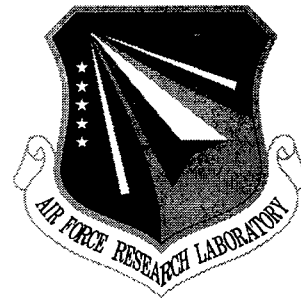


AFRL-SN-RS-TR-2001-79
Final Technical Report
May 2001



REVOLUTIONARY ADVANCES IN UBIQUITIOUS, REAL-TIME MULTICOMPUTERS AND RUNTIME ENVIRONMENTS

Mississippi State University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E339

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

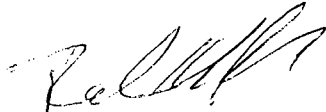
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

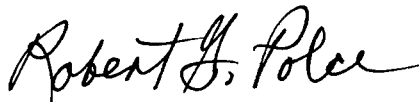
20010713 063

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-SN-RS-TR-2001-79 has been reviewed and is approved for publication.



APPROVED: RALPH KOHLER
Project Engineer



FOR THE DIRECTOR: ROBERT G. POLCE
Chief, Rome Operations Office
Sensors Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/SNRT, 26 Electronic Pky, Rome, NY 13441-4514. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REVOLUTIONARY ADVANCES IN UBIQUITIOUS, REAL-TIME
MULTICOMPUTERS AND RUNTIME ENVIRONMENTS

Ashok K. Agrawala

Contractor: Mississippi State University
Contract Number: F30602-96-1-0329
Effective Date of Contract: 26 August 1996
Contract Expiration Date: 31 December 1999

Short Title of Work: Revolutionary Advances in Ubiquitous, Real-
Time Multicomputers and Runtime
Environments
Period of Work Covered: Aug 96 – Dec 99

Principal Investigator: Anthony Skjellum
Phone: (601) 325-8435
AFRL Project Engineer: Ralph Kohler
Phone: (315) 330-2016

Approved for public release; distribution unlimited. .

This research was supported by the Defense Advanced Research
Projects Agency of the Department of Defense and was monitored
by Ralph Kohler, AFRL/SNRT, 26 Electronic Pky, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE May 2001		3. REPORT TYPE AND DATES COVERED Final Aug 96 - Dec 99
4. TITLE AND SUBTITLE REVOLUTIONARY ADVANCES IN UBIQUITIOUS, REAL-TIME MULTICOMPUTERS AND RUNTIME ENVIRONMENTS			5. FUNDING NUMBERS C - F30602-96-1-0329 PE - 62301E PR - D985 TA - 01 WU - 10	
6. AUTHOR(S) Ashok K. Agrawala				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Mississippi State University Sponsored Programs Administration PO Box 6156, 305 Bowen Hall Mississippi State, MS 39762			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/SNRT 3701 North Fairfax Drive 26 Electronic Pky Arlington, VA 22203-1714 Rome, NY 13441-4514			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-SN-RS-TR-2001-79	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Ralph Kohler, SNRT, 315-330-2016				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE N/A	
13. ABSTRACT (Maximum 200 words) This work was a grant to enhance the Maruti operating system in several ways, in order to provide Mississippi State with a platform upon which their work on the Real-Time Message Passing Interface could be developed. Key technical achievements: (1) Developed predictable Myrinet communications for use in a real-time NOW; (2) Developed the MSU-Kernel to provide a POSIX OS for real-time NOWs; (3) Developed and implemented an algorithm for deploying a globally synchronized clock in a real-time NOW; (4) Developed an improved real-time scheduler for the Maruti hard real-time operating system at University of Maryland (UMD); and (5) Introduced a new parametric approach in Maruti for dynamic scheduling at UMD. Details of the results of the work are presented in papers, thesis and project reports.				
14. SUBJECT TERMS Maruti, POSIX OS, High Performance Zero-sided Messaging, Real-Time Scheduler, Dynamic Scheduling			15. NUMBER OF PAGES 230	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

TABLE OF CONTENTS

PARA	TITLE	PAGE
1	Technical Status Report	1
1.1	Key Technical Achievements	1
1.2	Key Publications	2
2	Business Report	3
	Paper: Time Based Linux for Real-Time NOWs and MPI/RT	4
	Figure 1: Message Latency	5
	Paper: A Fine-Grain Clock Synchronization Mechanism for QoS Based Communication on Myrinet	6
	Figure 1: Myrinet network showing NIC and components	7
	Figure 2: Clock message packet	11
	Figure 3: Clocks assessed	11
	Figure 4: Deviation from Master Clock (Nanoseconds) for all slaves	17
	Table 1: Measured drift between hardware clocks of slaves and master	18
	Table 2: Clock error and overhead for various synchronization periods	19
	Paper: A Synchronized Real-Time Linux Based Myrinet Cluster for Deterministic High Performance Computing and MPI/RT	22
	Figure 1: Predictable message passing latency in BDM-RT	26
	Figure 2: Time-line messaging	29
	Figure 3: Jitter in waiting time at Lanai	30
	Figure 4: Jitter in message receive time	31
	Paper: Predictability and Performance Factors Influencing the Design of Real-Time Messaging Layers	33
	Paper: A Real-Time Message Layer over Myrinet Networks	137
	Paper: Final Report from the University of Maryland for DARPA Contract 996144601	182
	Figure 1: Screen Display of MAGIC	187
	Figure 2: Using a single APIC timer interrupt to dispatch real-time tasks	189
	Figure 3: Dispatching time accuracy using single APIC interrupt	191
	Figure 4: Using two APIC interrupts to dispatch real-time tasks	192

Figure 5: Setting up the APIC timer for double-interrupts	194
Figure 6: The interrupt service routine for the APIC timer	195
Figure 7: Accuracy of dispatching time using double APIC interrupts	196
Figure 8: Frequency histogram of the err Kernel	197
Figure 9: Frequency histogram of the err Task	198
Figure 10: Static Cyclic Scheduling	201
Figure 11: Parametric Calendar Structure	202
Figure 12: Maruti Application Life Phases	203
Figure 13: Operaton Model of the dynamic time-based scheduling system	205
Figure 14: Constraint Graph for $\Gamma^{1,2}$	209
Figure 15: Elimination of f_2^2 and c_2^2 from $\Gamma^{1,2}$	211
Figure 16: Dependency Graph	214

1 Technical Status Report

The project achieved all key goals. This effort has accomplished work to show how to create real-time network of workstations (NOWs).

1.1 Key technical achievements

1. Developed predictable Myrinet communication for use in a real-time NOW.
 - Thesis: Predictability and Performance factors influencing the design of real-time messaging layers. S. Chakravarthi, MS. Thesis, Mississippi State University, 2000.
 - Achieved message transfer latencies of 30 μ s without requiring initial handshaking.
 - Developed Myrinet drivers tailored to provide deterministic DMA latencies to improve real-time performance.
 - Developed 1-copy message transfer using shared memory mapped to the Myrinet NIC.
2. Developed the MSU-Kernel to provide a POSIX OS for real-time NOWs. The achievements include:
 - Published paper: Time-based Linux for Real-Time NOWs and MPI/RT. M. Apte, S. Chakravarti, A. Pillai, A. Skjellum, and X. Zan. In IEEE Real-Time Systems Symposium, Phoenix AZ, Dec 1999.
 - Developed a real-time scheduler that supports non-real-time tasks, hard real-time tasks, and greedy real-time tasks.
 - Utilized the processor's APIC timer to reduce scheduling granularity to 30ns.
 - Achieved scheduling overhead of no more than 10m s for 8K-byte messages.
 - Achieved message latency jitter of no more than 30ms.
 - Project report: A Real-Time Message Layer over Myrinet Networks. X. Zan, MS. Project Report, Mississippi State University, 2000.
3. Developed and implemented an algorithm for deploying a globally synchronized clock in a real-time NOW.
 - Paper: A Fine-Grain Clock Synchronization Mechanism for QoS Based Communication on Myrinet. S. Chakravarthi, A. Pillai, J. Padmanabhan, M. Apte, and A. Skjellum. . Submitted to the 21st International Conference on Distributed Computing Systems (ICDCS-21). <http://www.cs.msstate.edu/~tony/documents/Message-Passing/ICDCS2001-GClock.pdf>

- Paper: A Synchronized Real-Time Linux Based Myrinet Cluster for Deterministic High Performance Computing and MPI/RT. M. Apte, S. Chakravarthi, J. Padmanabhan, and A. Skjellum. Submitted to the Ninth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2001). <http://www.cs.msstate.edu/~tony/documents/Message-Passing/WPDRTS-2001.pdf>
 - Devised an algorithm to synchronize clock in a NOW by sending periodic, low-overhead Myrinet messages with predictable latency over a single Myrinet switch.
 - The CPUs were synchronized to within $\pm 5\mu s$ of each other.
 - Demonstrated high-performance predictable zero-sided messaging using global schedules
4. Developed an improved real-time scheduler for the Maruti hard real-time operating system at University of Maryland (UMD).
 - Utilized the APIC countdown timer to improve the accuracy of dispatching.
 - Developed a “double-interrupt” mechanism to pre-load the level-1 cache in order to reduce the variability of dispatch times.
 5. Introduced a new parametric approach in Maruti for dynamic scheduling at UMD.
 - This scheduling technique allows the specification of a wide variety of timing constraints (e.g., ready time, deadline, communication constraint, mutual exclusion, together, and relative timing).
 - The dynamic scheduling capability of this approach allows the ability to add aperiodic tasks at runtime and to modify scheduling decisions at runtime depending on the characteristics of executing tasks.

1.2 Key Publications

Details regarding the results of this work are presented in the papers, thesis, and project reports attached with this report.

2 Business Report

2.1 Expenses: (4/1/96 – 12/31/99)

Mississippi State University	
Salary	393,922.53
Wages	34,376.61
Fringes	86,105.66
Travel	72,989.53
Subcontract	446,150.00
Contractual	44,028.36
Commodities	1,616.33
Equipment	58,906.98
Overhead	225,280.00
Total	\$1,363,376.00

University of Maryland	
Salary and Wages	228,195.62
Fringes	42,242.25
Travel	8,863.19
Commodities	8,608.99
Equipment	15,219.00
Other Direct Costs	4,824.14
Overhead ²	138,196.81
Total	\$446,150.00

Time Based Linux for Real-Time NOWs and MPI/RT

Manoj Apte, Srigurunath Chakravarthi,
Anand Pillai,
Anthony Skjellum, Xin Yan Zan
Department of Computer Science and
NSF Engineering Research Center
Mississippi State University. MS 39762
{manoj,ecap,anand,tony,xyz}@erc.msstate.edu

Abstract

The Real-Time Message Passing Interface (MPI/RT) is a communication layer middleware standard that is aimed at providing guaranteed Quality of Service for data transfers on high performance networks. It poses "middleout" requirements both on applications and on the operating system. In this paper, we consider the "middledown" issues by modifying a POSIX compliant operating system in order to support hard real time scheduling, a feature needed for efficient MPI/RT on real-time Networks of Workstations (NOWs). This paper describes the evolution of Turtle, a variant of RT-Linux, that will later support a prototype implementation of time-driven MPI/RT. Analysis, design approach, and results for Turtle are discussed.

1. Introduction

Commodity desktop machines are now powerful enough to handle several tasks concurrently through time sharing. It is feasible to use such workstations for performing Quality-of-Service (QoS) sensitive tasks while still allowing interactive tasks in the foreground. Integrated with a high-performance network supporting real-time protocols, one can also envision real-time Networks of Workstations (NOWs) used for distributed real-time computing, while serving as standard desktops. This paper describes the design of Turtle, a time-based operating system to manage synchronized real-time clusters of workstations.

MPI/RT is an emerging real-time message passing standard, that may become the de-facto standard for high performance distributed real-time applications[3]. The standard prescribes the design for a middle-ware message passing layer to support message passing

with guaranteed Quality of Service (QoS). The MPI/RT programming model is fairly distinct from the current practices in embedded and distributed real-time system design in being much more conducive to portability and providing hard real-time guarantees based on priorities, events and time. This poses several new requirements on the features provided by the underlying real-time operating system, which we aim to satisfy through Turtle.

There are several real-time operating systems available commercially. VxWorks, pSOS, QNX, LynxOS and RT-Linux [1] are priority based systems. Microsoft's RealTo, RT-Mach [2], MARUTI and MARS are examples of systems scheduling tasks based on time. The RT-Mach resource reservation model comes closest to satisfying the requirements of MPI/RT. Turtle incorporates a hybrid form of the RT-Mach reservation model and the Rate Controlling model by Yau and Lam [4] into a cluster of Myrinet [5] based Linux workstations. Existing messaging layers over Myrinet can not, in general, provide predictable-latency message transfer in the presence of shared resources such as the PCI bus, RAM, and network switches. Our RT messaging sub-system provides bounded end-to-end message latency by eliminating resource contention. The messaging layer includes channel, QoS, and buffer abstractions which directly correspond to those in MPI/RT, to ensure a highly optimized middle-ware library.

2. Design

The current laboratory setup consists of 2 clusters of dual-capable 200 MHz Pentium Pro machines with 128MB RAM, 2 GB SCSI disk, and Myrinet network using the LANai 4.0 cards. The Turtle system is based on the Linux 2.0.30 and RT-Linux 0.5. The Pentium APIC timer in one-shot mode is used for scheduling.

2.1 Turtle Scheduler

The Turtle scheduler incorporates a novel real-time scheduler based on the Rate controlled scheduling model by Yau and Lam [4], and the RT-Mach reservation model [2]. The scheduler is designed to allow concurrent execution of real-time tasks, and Linux processes which are run as best-effort. The Linux kernel is guaranteed a minimum QoS to ensure the workstation allows interactive usage.

Turtle provides a matching kernel level API for MPI/RT time-driven channels. All hard real-time tasks request QoS in terms of start and end time, period, computation time, and the deadline with respect to start of the period. The QoS is a contract between the task and the scheduler. Hence, once a task is given its requested computation time, its critical deadline is

updated to the next period. The scheduler prioritizes tasks based on critical deadlines, thus ensuring that a misbehaving task does not affect QoS guarantees for other tasks. Such temporal isolation is critical for real-time scheduling on COTS systems that show non-deterministic behavior.

In addition to its requested QoS, the Linux kernel is treated as a special greedy task, that is given extra computation time as long as there are no other READY tasks. Efforts are underway to support more than one greedy real-time tasks that may request extra computation time as a multiple of some optional time quantum during slack time.

2.2 Myrinet Real-Time Messaging Layer

The LANai network co-processor is configured with a specialized Myrinet Control Program (MCP) adapted for QoS-sensitive behavior. PCI bus contention from non-real-time traffic is eliminated by performing "blocking" PCI DMA, i.e. disallowing potential PCI bus contenders from being scheduled on the CPU during the transfer.

The Turtle cluster maintains a globally synchronized master-slave clock over Myrinet that enables high performance parallel applications to take advantage of synchronized persistent communications. It also enables the scheduling algorithm to pre-plan data transfers to avoid contention at the switches, thus ensuring consistent high bandwidth.

The Myrinet messaging layer provides an interface consistent with the MPI/RT standard.

3. Results

The Turtle scheduler improves interrupt latency, by emulating the 8254 timer for Linux. The scheduling overhead is maintained between 4 and 16 μ s, with an average of 6.5 μ s. The system was tested for up to 12 real-time tasks without failure. It can support a task period as low as 70 μ s.

Guaranteed message passing latency is achieved by the "blocking" PCI DMA transfer. Eg. A 2KB data transfer from the host memory to network buffers is bound by 20 μ s. No such hard upper-bound can be established for a traditional non-blocking DMA.

Maximum error in the slave clocks is bound within $\pm 4\mu$ s. With this set-up we measured message passing latency for 12 bytes messages to be guaranteed between 6 and 13 μ s as shown in Figure 1.

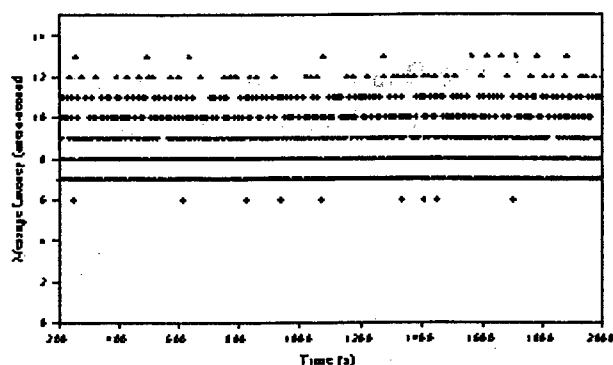


Figure 1: Message Latency.

4. Conclusions

The Turtle system is well suited as the base for a high performance implementation of the MPI/RT 1.0 library. MPI/RT channels can be directly abstracted as real-time tasks with an equivalent QoS. The system takes advantage of the time-driven paradigm to eliminate the need for explicit synchronization primitives and the associated overheads. Policing of PCI DMA transfers avoids contention with non-real-time traffic. Turtle maintains a fine-grained global clock using real time tasks. Resource contentions in the messaging system are avoided by synchronizing over the global clock.

The project also demonstrates the feasibility of using commercial desktop systems for hard real-time computation. The experience gained will help identify bottlenecks in current commodity hardware that are non-conducive to real-time applications.

5. References

- [1] M. Barbanov and V. Yodaikin, "Real-time linux", Linux Journal, March 1996.
- [2] C. Lee, R. Rajkumar, and C. Mercer. "Experiences with processor reservation and dynamic QoS in real-time Mach", In proceedings of Multimedia Japan 96, April 1996.
- [3] MPI/RT: Real-Time Message Passing Interface Standard 1.0. <<http://www.mpirt.org>>.
- [4] David K. Y. Yau and Simon S. Lam, "Adaptive Rate-Controlled Scheduling for Multimedia Applications", In proceedings of ACM Multimedia Conference, Boston MA, November 1996.
- [5] Myricom Inc. LANai3/4 Documentation, 1997. <<http://www.myri.com/scs/L3/documentation.html>>.

A Fine-Grain Clock Synchronization Mechanism for QoS Based Communication on Myrinet¹

Srigurunath Chakravarthi, Anand Pillai, Jothi Padmanabhan,

Manoj Apte, and Anthony Skjellum².

High Performance Computing Laboratory,

Department of Computer Science ,

Mississippi State University

{ecap,anand,jothi,manoj,tony}@hpcl.cs.msstate.edu

Abstract

Clock synchronization is a fundamental requirement for any real-time distributed system operating with global schedules. This paper describes the design and implementation of a high accuracy ($\pm 4 \mu s$) global clock on a Myrinet [4] gigabit/s system area network of PCs with considerably low software overheads. The global clock is based on a master-slave internal clock synchronization scheme [15]. A novel approach has been adopted to improve synchronization accuracy. The programmability of the Myrinet interface card and the presence of an on-board Real Time Clock [4] have been utilized to counter the undesirable effects of unpredictability in the latency of clock messages. The resulting synchronization facilitates global scheduling of distributed real-time tasks, and provides a framework to build support for Quality of Service in distributed high-performance environments.

1 Introduction

Coscheduling and minimization of network contention are key to high performance cluster computing. Coscheduling is possible if a global clock is maintained across all nodes in a cluster. Maintaining such a global clock is a nontrivial task, since variation in temperature and pressure changes the natural frequency

¹ This work was funded in part by DARPA, US Navy TASP, and by the National Science Foundation.

² Corresponding Author. Phone: 662 325 8435. Fax: 662 325 8997. Email : tony@hpcl.cs.msstate.edu

of crystal-based oscillators. The difference in clock frequencies of individual nodes is termed drift. The resulting difference in the absolute clock values is called skew. A clock synchronization mechanism must correctly estimate the drift and skew of the local clocks to maintain a virtual global clock. This paper describes the design and implementation of a high-accuracy internal clock synchronization mechanism on the high-speed Myrinet [4] network. Internal clock synchronization [11] is synchronization of clocks with respect to one another but not with respect to the outside world. It is achieved by using a master-slave mechanism.

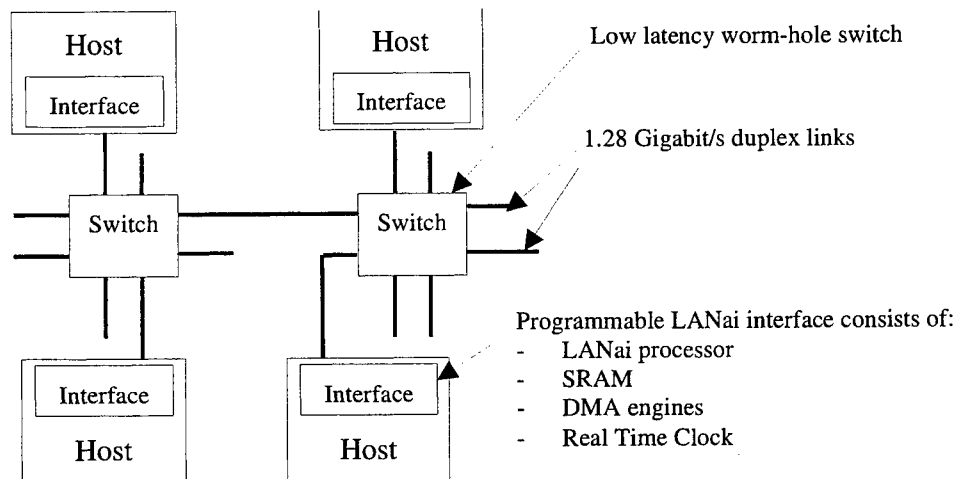


Figure 1: Myrinet network showing NIC and components

Myrinet is a gigabit/sec system area networking technology that is widely used in distributed high-performance computing. Figure 1 shows the key components of a Myrinet network: 1. A programmable Network Interface Controller (NIC) with the LANai processor, SRAM, and a Real Time Clock (RTC), and 2. Source routed cut-through Myrinet switches. So far, Myrinet has been primarily used for high-performance communication. Popular messaging layers such as GM [12], FM [13] and BIP [14] are designed for low-latency and high-bandwidth, with no particular support for Quality of Service or real-time communication. FM QoS [6] is the only known messaging layer providing predictably low latencies between network interfaces across a Myrinet network of which we are aware.

Myrinet worm-hole switches have predictably low fall-through latencies of the order of 500 ns [4]. However, the presence of contending traffic at a switch can cause unacceptably high jitters in latency [6]. Thus the key to enabling predictable communication across a worm-hole routed network such as Myrinet lies in generating conflict-free schedules. This directly necessitates building a global view of time across the network. This global view of time has to be sufficiently fine-grained and accurate to facilitate useful utilization of Myrinet's high bandwidth and low latency. Further, the mechanism that implements the global view of time should have acceptably low overheads to achieve acceptable performance.

This paper describes a global clock implementation in software that meets the above requirements. The implemented global clock is already being used as a vital component of BDM-RT [5] – a low-level real-time Myrinet messaging layer. The key contribution is that this high-accuracy clock can pave the way for QoS based communication with predictable messaging latencies over Myrinet.

The rest of the paper is organized as follows: Section 2 presents the problem of clock synchronization and viable solutions in high-performance distributed environments. Section 3 describes our approach and design. Section 4 presents results and analysis of these results on a 8-node, 1-switch Myrinet network . Section 5 concludes the paper and presents planned improvements and future work.

2 Synchronization in high-performance environments

Algorithms that solve the Byzantine general's problem [11] require communication that grows as $\theta(n^2)$ with respect to the number of clocks involved in the synchronization. Hence, these algorithms do not scale well with the network size and can in general be categorized as having unacceptably high communication overhead for use in high-performance environments. Hardware based clock synchronization using GPS systems [16] is a viable alternative, but is usually too expensive specially for large networks.

In a Myrinet based the network of PCs, the problem translates to synchronizing either the host clocks or the on-board LANai Real Time Clocks (RTC). Synchronization of LANai RTCs helps in generating conflict-free schedules for the link traffic. By conflict free, we mean that no packet will be scheduled to arrive at a

Myrinet switch (see Figure 1) with the same outgoing port as another packet that is currently falling through the switch, thus eliminating any waiting time at the switch. FM QoS achieves synchronization at the Myrinet interface by building a global view of time using network feedback. While this mechanism enables conflict-free scheduling of messages at the network interface, synchronization is still absent at the host level. That is, the resulting synchronization can not by itself be used to implement global CPU schedules for host-to-host synchronization nor to guarantee end-to-end QoS.

An alternate synchronization scheme is to synchronize the host clocks (on which CPU schedules are based) instead of the on-board clocks at the network interface. This latter mechanism mainly facilitates the implementation of global CPU schedules for distributed tasks, and can also ensure conflict-free link schedules if protocol processing delays are predictable. In other words, if the time spent by a message at the sender's node before being sent on the network is predictable, the host clock can itself be used to schedule link traffic.

We chose to implement host-level synchronization mainly because of the following advantages: 1. It facilitates global scheduling of real-time channel tasks [10]. 2. On our platform, host clocks are more fine-grained (32-bit 5 ns counters) than the on-board RTCs (32-bit 500 ns counters). 3. We have developed a real-time messaging layer BDM-RT [5] that meets the above-mentioned protocol processing requirements to ensure conflict free link schedules.

3 Our Design

Our algorithm is based on a master-slave synchronization scheme. One node in the network is designated as the master node and the rest as slave nodes. The master periodically broadcasts its clock value to each of the slaves. Each slave constructs a virtual clock based on the received master clock values. This scheme scales well because the number of clock message transfers grows as $\theta(n)$ with the number of nodes in the network. Additionally, the relatively low bandwidth and processing time required for this algorithm is suitable for high-performance environments.

3.1 Integrity of clock messages

It is well known that the accuracy of slave clocks is limited by the variation in the time taken to transfer clock messages [7]. More specifically, if the latencies of individual clock messages are bounded by T_{\min} and T_{\max} , there is potentially an error of $(T_{\max} - T_{\min})$ units in the master's clock value accessed by a slave. That is, the master's clock reading, T_{master} , at the instant usable by the slave can be represented as:

$$T_{\text{master}} = T_m + T_{\min} + \Delta \quad (1)$$

Where

T_m is the master clock's time-stamp carried by the clock message,

T_{\min} is the theoretical minimum latency incurred by the clock message,

$0 \leq \Delta \leq (T_{\max} - T_{\min})$, and

T_{\max} is the theoretical maximum latency incurred by the clock message

The inaccuracy of the slave clock's reading of the master clock is therefore at least Δ units of time. Studies conducted by us on the predictability of host-to-host latency on a Myrinet network [1] revealed that latency variation (jitter) is rather high compared to the absolute latency of messages on high-performance messaging layers. This is largely attributed to the unpredictable data transfer time between the host and LANai memory across a shared bus. For example, using periodic messages of length 32 bytes with BDM, our non real-time messaging library on Linux [9], jitters as high as 300 μsec (with no guaranteed bound) were observed [5][9]. This is rather high compared to the average latency of 30 μsec .

The key to improving global clock accuracy in a master-slave scheme thus lies in improving the accuracy of remote clock (master clock) accesses. One way of doing this is to improve the predictability of clock message latencies, thereby reducing Δ . This method is rather difficult to implement on commodity platforms given the various extraneous factors (such as contention of shared resources, scheduling delays, variation in traffic load, etc.) influencing the predictability of end-to-end latency. An alternate solution is to bound, if possible, the value of Δ with relatively higher accuracy than the variation of Δ itself. The next

sub-section describes how Δ can be estimated with reasonably high accuracy on any Myrinet messaging sub-system, and discusses our implementation.

3.2 Estimation of message latency

The actual delay incurred by a clock message is measured by the messaging subsystem and is made available to the slave at the time of receipt of the clock message. By recording delays at the various stages of protocol processing, the master clock is made to “keep ticking” while in transit to the slaves. Figure 2 is

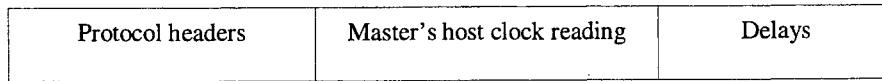


Figure 2 Clock message packet

a schematic representation of the clock message sent by the master at every re-synchronization cycle.

Barring the protocol headers, clock messages mainly contains two fields of information: The master's clock value (T_m), and the protocol processing delays encountered by the clock message (D_m).

To understand how the delays are recorded, the typical stages involved in the transfer of messages on a Myrinet network are summarized here. On the sender's side, these constitute: 1. Protocol processing at the host, 2. Data transfer between host and LANai on-board buffers, 3. Protocol processing by the Myrinet Control Program (MCP) at the network interface, and 4. Network DMA of message to remote network interface. On the receiver's side, the stages are as follows : 1. Network DMA to receive a message, 2. Protocol processing and buffering of received message, 3. Transfer of message to host memory, and 4. Protocol processing by the host library.

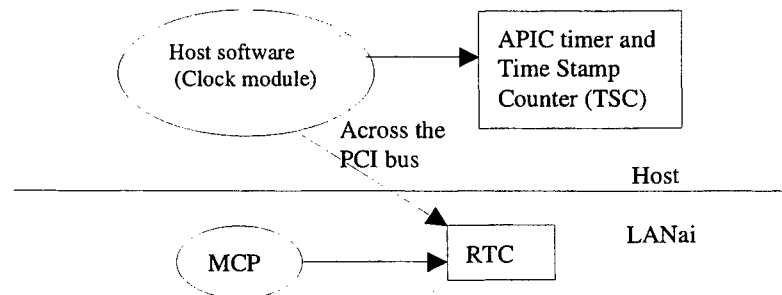


Figure 3: Clocks Accessed

We now describe how to compute L_n – the network latency. In the absence of any extraneous traffic during the broadcast of clock messages, L_n equals the total switch fall-through time. Assuming S as the number of switches traversed by a clock message, and a constant fall-through time, L_{switch} , of 0.5 usec [4], the above equation can be rewritten as follows:

$$T_{\text{master}} = T_m + D_m + (T_{\text{sn2}} - T_{\text{sn1}}) + (S * L_{\text{switch}}) \quad (3)$$

Our current implementation ensures that conflicting Myrinet traffic is not scheduled during clock re-synchronization periods. Hence we use equation 3 to calculate T_{master} . Because of this simplification, an additional inaccuracy is introduced in the measurement of T_{sn1} . T_{sn1} should ideally be the RTC timestamp when a clock message shows up at the slave's LANai interface. However, the slave MCP typically runs in an infinite loop and checks for messages ready to be received only periodically in its activity loop. Because of this polling delay, a certain amount of time may have elapsed before the MCP detects that a message is ready to be received. This elapsed time cannot be accounted for using Equation 3.

Equation 2 can be used to eliminate this inaccuracy, and to operate without any restrictions on conflicting Myrinet traffic. L_n can be calculated by the master node's MCP by measuring the actual amount of time it takes to DMA a clock message. This will include delays at switches, and any polling delays at the slave node. Implementation of this will require an extra message to transfer the value of L_n from the master to the slave. This is envisioned as future work.

3.3 Slave virtual clock algorithm

Upon calculation of the master's clock value, the slave updates its virtual clock based on the current virtual clock value and the computed master's clock value. The algorithm used to compute the slave's virtual clock to be used until the next re-synchronization cycle is briefly described in this section. One of the primary goals that governed the choice of our algorithm was minimizing the computation overhead. The algorithm is also tolerant to faults arising from missed clock messages.

The availability of the on-board Real Time Clock (RTC) at the Myrinet interface is crucial to the ability to record processing delays. Additionally, the RTC can be memory-mapped to host memory, making it accessible directly from the host. Figure 3 illustrates this. Let us denote the master's host-clock with C_{mh} , the master's RTC at the network interface with C_{mn} , the slave's host-clock with C_{sh} , and the slave's RTC with C_{sn} . At every re-synchronization cycle, the master clock task reads off C_{mh} and C_{mn} consecutively with no intervening delays. Let this time be T_{mh} by C_{mh} and T_{mn1} by C_{mn} .

The clock message is constructed by the host library with the field T_m equal to T_{mh} , and D_m equal to T_{mn1} . After all protocol processing at the master has been done, the sender's Myrinet Control Protocol (MCP) is ready to DMA the message on the network. At this moment, let the C_{mn} reading be T_{mn2} . The field D_m is overwritten with $(T_{mn2} - T_{mn1})$, i.e. the sum of all delays incurred at the master node.

At the slave node, the MCP reads off C_{sn} at the beginning of receipt of every message. Let this value be T_{sn1} for the clock message. The clock message is fully received and stored on LANai SRAM until the slave clock task is ready to pull it off the buffer. After the slave task has copied the clock message to host memory, it reads off C_{sh} and C_{sn} in succession with no intervening instructions. Let these timestamps be T_{slave} and T_{sn2} . The master's clock value at this instant (i.e., when C_{sh} reads T_{slave}) can be calculated as follows:

$$T_{master} = T_m + D_m + (T_{sn2} - T_{sn1}) + L_n \quad (2)$$

Where L_n is the network latency incurred by the clock message. The value of T_{master} computed as above is typically inaccurate because of inaccuracies in the measurement of T_{mn1} (on which D_m depends) and T_{sn2} . As described above, both T_{mn1} and T_{sn2} involve reading the value on the on-board RTC from the host processor. This operation occurs across the PCI bus and inherently induces an inaccuracy (empirically measured to be approximately 2 μs on our architecture testbed).

For simplicity let us assume that each slave constructs a virtual clock by assuming a constant drift rate between the master and its clock between consecutive re-synchronization cycles. At any instant in the n^{th} re-synchronization interval (i.e., between the n^{th} and the $(n+1)^{\text{th}}$ clock message broadcasts), the virtual clock at a slave can be extrapolated as:

$$T_v = C + \delta_n * (T_{sh} - T_{sh}^n) - \epsilon * (T_{sh} - T_{sh}^n) / P_s \quad (4)$$

Where

T_v is the value of the synchronized virtual clock at a slave,

C is a constant that corrects the boot-up time offset between the master and slave clock,

δ_n is the computed relative drift between hardware clocks of a slave and the master for that period,

T_{sh} is the reading of the slave's hardware host clock,

P_s is the synchronization period in slave time units,

T_{sh}^n is the reading of the slave's hardware host clock at the time of the n^{th} re-synchronization, and

ϵ is the error in the synchronized virtual time at the time of the n^{th} re-synchronization.

The second expression on the right side of the above equation corrects for the difference in clock frequencies of the master and slave clocks. The third expression amortizes the error at the previous synchronization over the subsequent synchronization period and ensures convergence of the synchronized virtual clock to the master clock.

C and δ_n are computed based on the master clock values received and the local slave clock reading at the time of receipt by the slave task. Let T_{master}^n represent the value of the master clock computed using equation 2 (or equation 3, as the implementation may be) for the n^{th} re-synchronization master clock message. Let the value of the slave hardware clock at the instant of receipt of the n^{th} master clock message be T_{slave}^n .

Then, C is found by computing the difference between the master and slave clock readings for the first synchronization cycle, and remains constant thereafter. That is,

$$C = (T_{\text{master}}^1 - T_{\text{slave}}^1) \quad (5)$$

δ_n is recalculated at every re-synchronization cycle based on the relative drifts of the master clock and the local slave clock.

$$\begin{aligned} \delta_n &= (T_{\text{master}}^n - T_{\text{master}}^{n-1}) / (T_{\text{slave}}^n - T_{\text{slave}}^{n-1}) \quad \text{for } n > 1 \\ \delta_1 &= 1 \end{aligned} \quad (6)$$

$\varepsilon = T_v^n - T_{\text{master}}^n$, where T_v^n is synchronized virtual time at the time of the n^{th} re-synchronization and extrapolated based on the clocks' state at the $(n-1)^{\text{th}}$ to n^{th} re-synchronization intervals.

In order to induce fault tolerance to missed clock messages at the slave and to decrease the undesirable effect of sporadic transmission errors on the virtual clock, the actual algorithm used to compute δ_n is slightly different as compared to the one described by equation 4.

Instead of assuming a constant drift rate between two consecutive re-synchronization intervals the drift rate is assumed constant and averaged over a longer sliding window of time. The number of synchronization periods the window spans is called the "window size." At each re-synchronization the window size increases by one. When the size of the window reaches a maximum, W_{max} , the windows slides forward such that the size of the window is now W_{min} . To maintain some history of drift the window is not allowed to slide forward completely and reach zero size. Relative drift, δ_n , is computed within this window as,

$$\delta_n = (T_{\text{master}}^n - T_{\text{master}}^o) / (T_{\text{slave}}^n - T_{\text{slave}}^o) \quad (7)$$

where,

T_{master}^o is the master clock time at the beginning of the current window, and

T_{slave}^o is the slave host time at the beginning of the current window.

The averaging smoothes out isolated erroneous clock messages and longer window sizes increase this smoothing effect. Clock behavior and hence the frequency changes over time depending on room temperature and other factors. If, instead of a sliding window, a stationary enlarging window is used or if a long window size is chosen, then the effect of isolated erroneous clock messages will affect the synchronized virtual clock longer, but to a monotonically diminishing extent. Further, recent changes in relative drift between master and slave will take longer to correct the synchronized virtual clock. Hence the parameters of the sliding window should be chosen carefully.

In our implementation the window does not slide continuously as to achieve that, the clock status at every re-synchronization needs to be stored in memory, which is an overhead.

4 Experiments, Results and analysis

4.1 Hardware and Software Configuration

All experiments have been performed on an eight node cluster connected by Myrinet. Each node consists of a Pentium Pro 200 MHz processor with Intel FX440 motherboard. A single 32 bit PCI bus (33 Mhz) is shared by Ethernet, Myrinet and SCSI interfaces. The Myrinet PCI interface consists of the LANai 4.x processor with 1MB of SRAM. The operating system used is Linux 2.0.30. The Myrinet network is driven by the BDM-RT messaging library. This configuration represents a typical high performance cluster composed of commercial-off-the-shelf (COTS) desktops. The newest Intel processors and Myrinet can utilize all the results shown here.

4.2 Synchronization Accuracy

The accuracy of clock synchronization can be computed by recording the deviation between virtual global clocks maintained by the slaves and the master clock at regular intervals. In our experimental setup, we measure the deviation between the master clock and the global clock at every resynchronization period.

This approach ensures capturing the worst case skew, since the global clock in each slave is expected to have maximum deviation from the master clock just before re-synchronization.

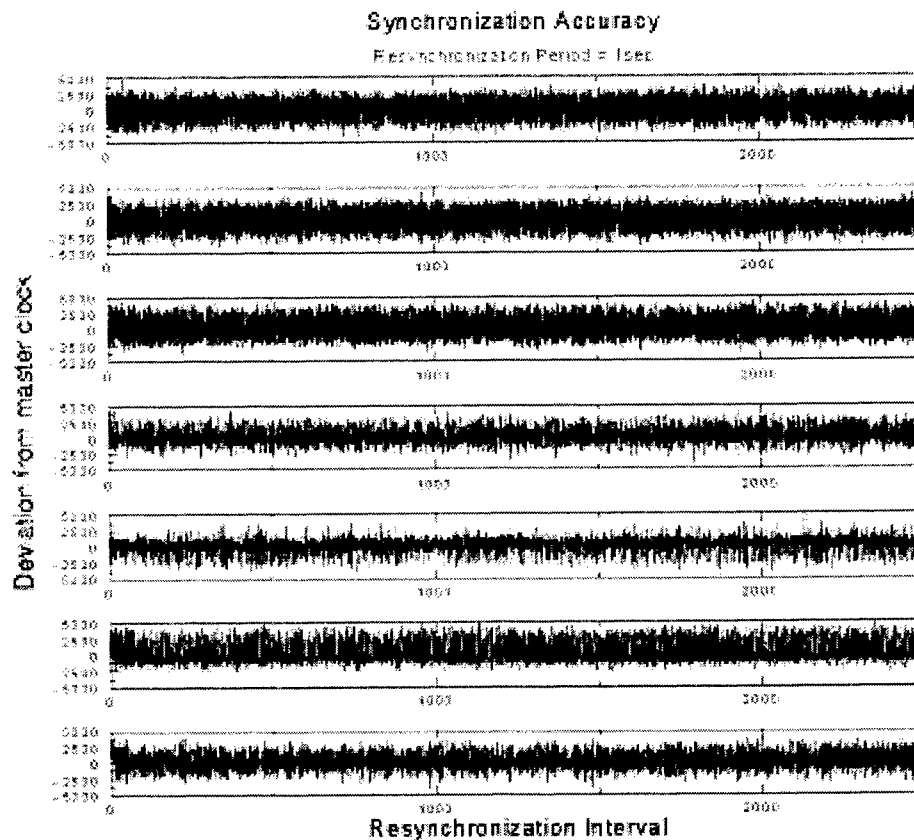


Figure 4: Deviation from Master Clock (Nanoseconds) for all slaves

The global clock module runs in each of the slaves at every resynchronization interval. It is instrumented to record the deviation between the global clock time and the master clock time computed using Equation 3 on the received master clock message, before executing the virtual clock algorithm (Equation 4). Apart from the deviation and the resynch interval count, other metrics such as the drift between the master and slave hardware clocks, and the time the clock message waited in the slave's LANai before being consumed by the host are also stored.

At this point, it is important to note that the deviation as measured above gives only an approximate value of the accuracy of the global clock. Ideally, accuracy should be computed using the deviation between

exact values of the global clock and the master clock. In our setup, we use an *estimated* value of the master clock, because of the unavailability of an external mechanism to access its exact value at the slave nodes. Hence, the measured accuracy is associated with an error equal to the error in the estimation of the master clock value at the slave.

Figure 4 shows a sample of the deviation for each of the 7 slave nodes over 2500 re-synchronization periods with a resynchronization interval of 1 second. All the global clocks are within $\pm 5\mu\text{sec}$ of the master clock at any instant of time. The table below shows the drifts between uncorrected hardware clocks of each slave with respect to the master. It is seen that our algorithm copes well with a range of drifts to ensure synchronization within $\pm 5\mu\text{sec}$ of the master.

Slave	Uncorrected Drift (ppm)
Slave 1	120
Slave 2	36
Slave 3	103
Slave 4	91
Slave 5	18
Slave 6	58
Slave 7	52

Table 1: Measured drift between hardware clocks of slaves and the master

4.3 Optimizing the resynchronization interval

One of the primary goals of our clock synchronization algorithms is to get maximum accuracy while incurring as low an overhead as possible. Increasing the resynchronization interval reduces the overhead on network and CPU resources. However, it is expected that increasing the resynchronization interval would result in a decrease in the accuracy of the global clock due to error accumulated from uncorrected drift.

The network overhead associated with the clock synchronization algorithm is measured as the ratio of the time spent on sending clock messages to the clock resynch period. The time spent by the master for broadcasting its timestamp was calculated by measuring the RTC timestamps just before and after the broadcast and taking their difference. Our experiments show that this time is about 32.5 μ s. The estimated maximum CPU time required is 80 μ s.

The following table shows the worst case deviation of the clocks and the corresponding network and CPU overheads for resynchronization periods of 100ms, 1s, 10sec, and 30sec.

Period	Minimum Deviation (ns)	Maximum Deviation (ns)	Network Overhead %	CPU Overhead %
100ms	-4831	4783	3.25×10^{-2}	8×10^{-2}
1sec	-4174	4969	3.25×10^{-3}	8×10^{-2}
10sec	-4583	5069	3.25×10^{-4}	8×10^{-2}
30sec	-8602	8835	1.08×10^{-4}	2.7×10^{-2}

Table 2: Clock error and overhead for various synchronization periods

It is observed that increasing the resynchronization period has no noticeable effect on the accuracy for the periods studied. Our reasoning for this behavior is that for the range of periods studied, the error accumulation due to uncorrected drift during the period is insignificantly small compared the error in estimation of the master clock value. In other words, the inaccuracy is primarily because of measurement errors in reading the exact master clock value and the estimates for the synchronization message.

4.4 Fault tolerance

As mentioned earlier, the current design and implementation is tolerant to faults arising from transmission errors and occasionally missed clock messages, because of the averaging scheme used at the slaves. The primary shortcoming of master-slave synchronization algorithms is the threat of failure of the master node. Although, our current implementation does not cope with outage of the master node, the design makes it

feasible to add this fault-tolerance. Certain features of the current design make it fairly straightforward to designate a slave node to take-over as the master when required. For example, it is transparent to the scheduler and operating system whether the current node is a master or slave because both these modules are scheduled as periodic tasks with identical task parameters. Also, the master and slave modules use identical MCP code, thus making it simple for a node to switch between the roles of a master and slave. The implementation of master fail-over is planned as future work.

5 Conclusion and future work

The need for a high-accuracy global clock for real-time messaging and coscheduling on a distributed system was seen as the motivation for the work presented in this paper. Our goal was to design and implement a high-accuracy, low-overhead internal global clock on the high-speed Myrinet network. This was achieved by using a master-slave algorithm with a novel technique of recording protocol-processing delays encountered by clock messages. The resulting design can be implemented on any Myrinet-based messaging layer.

We achieved $\pm 5\text{usec}$ accuracy in an 8-node Myrinet cluster, incurring CPU and bandwidth overheads in the range of 0.03% to 0.07%. The algorithm is tolerant to transmission faults and missed clock messages. Further design improvements are planned to introduce master node fail-over and to reduce the incurred overhead. It is also planned to verify the exact accuracy of our global clock by using an external GPS based clock, and to further improve the accuracy by implementing the more exact scheme (Equation 2). It is envisioned that this low-overhead clock synchronization will pave the way for QoS based high-performance Myrinet messaging layers, and will facilitate fine-grain global synchronization of distributed tasks.

References

- [1] M.Apte, S. Chakravarthi, A. Pillai, A. Skjellum, and X. Zan. "Time-based Linux for real-time NOWs and MPI/RT" in *Proceedings of the Real-Time Systems Symposium*, 1999
- [2] M.Barbanov, and V. Yodaiken. "Introducing real-time linux", *Linux Journal*, pp19-23, Feb 1995
- [3] R.A.Bhoedjang, T. Ruhl, and H. E. Bal. "User-level network interface protocols", *IEEE Computer* 31, pp 53-60, Nov 1998.
- [4] N.J.Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. "A Gigabit-per-second Local Area Network", *IEEE Micro* 15, pp 29-36, Feb 1995.
- [5] S. Chakravarthi. "Predictability and Performance factors influencing the design of real-time messaging layers", *Master's thesis, Mississippi State University*, 2000.
- [6] K.Connelly, and A. Chien.. "FM-QoS: Real-time communication using self-synchronizing schedules" in *Proceedings of Supercomputing*, 1998.
- [7] F. Cristian. "Probabilistic clock synchronization", *Distributed Computing*, vol. 3, pp. 146-158, 1989
- [8] R. Gusella, and S. Zatti. "The accuracy of clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD", *IEEE Transactions on Software Engineering*, Vol 15, pp. 847-53, 1989.
- [9] G. Henley, , N. Doss, T. McMahon, and A. Skjellum. "BDM: A multiprotocol Myrinet Control Program and host application programmer interface", *Technical Report MSU-EIRS-ERC-97-3, Mississippi State University*, 1997.
- [10] A.Kanevsky, A.Skjellum and J.Watts. "Standardization of a communication middleware for high-performance real-time systems", in *Proceedings of the Real-Time Systems Symposium*, 1997.
- [11] L.Lamport, R. Shostak and M. Pease. "The Byzantine Generals problem", *ACM Trans. Prog. Lang. System* 4,3, pp 382-401, July 1982
- [12] Myricom, Inc. 1998. GM. http://www.myri.com/GM/doc/gm_toc.html. Last Accessed: July 20, 2000.
- [13] S.Pakin , M. Lauira, and A. Chien. "Illinois fast messages (FM) for Myrinet", in *Supercomputing*, 1995.
- [14] L.Prylli. BIP user reference manual. *Technical Report TR97-02, LIP/ENS-LYON, University of Lyon*, 1995.
- [15] J.A. Stankovic, and K. Ramamritham. "Clock Synchronization", *Advances in real-time systems*, pp 503, 1993.
- [16] B. Sterzbach. "GPS-based Clock Synchronisation in a Mobile, Distributed Real-Time System", *Real-Time Systems*, Vol.12, No. 1, 1997.

A Synchronized Real-Time Linux Based Myrinet Cluster for Deterministic High Performance Computing and MPI/RT*

Manoj Apte, Srigurunath Chakravarthi,
Jothi Padmanabhan and Anthony Skjellum[#]

*High Performance Computing Laboratory,
Dept. of Computer Science,
Mississippi State University.
{manoj,ecap,jothi,tony}@hpcl.cs.msstate.edu*

Abstract

This paper describes the design and implementation of a real-time cluster of PCs that provides globally synchronized scheduling and predictable messaging passing. A high-accuracy, fine-grain global clock implementation has been closely coupled with a time-based scheduler to facilitate finely synchronized global scheduling across the cluster. A real-time messaging layer provides predictable communication latencies over the interconnecting Myrinet network. This system provides a solid framework for QoS based real-time communication, and in particular facilitates efficient layering of high-performance real-time distributed middleware such as MPI/RT. We present experiments and results to demonstrate the degree of predictability and synchronization achieved on an 8-node Myrinet cluster.

1 Introduction

It is well known that clusters of commodity workstations are becoming a popular low cost alternative to super-computers for high-performance distributed computing [3]. Such systems also offer a higher degree of reliability through replication, better scalability, and portability. As distributed algorithms expand their scope to online computational simulation of complex phenomenon, data acquisition and analysis, telecommunications, multimedia and avionics, a new requirement is seen to emerge: The correctness of such applications depends not only on the result, but also their timeliness. To ensure correct operation, such real-time applications impose certain end-to-end Quality of Service (QoS) requirements on message delivery. These requirements can only be satisfied by appropriate design of the underlying architecture, operating system and middleware.

High performance distributed systems are not necessarily deterministic in their behavior. Most performance enhancements are aimed at improving the average case, resulting in a large gap between average and the worst case. A system required to provide precise end-to-end QoS guarantees is hence unable to take advantage of such enhancements. Determinism for guaranteed QoS and high performance are thus often complementary requirements. The system must be able to provide the application with an appropriate balance of determinism and average performance as dictated by its QoS requirements.

To provide temporal guarantees, individual tasks of a distributed real-time application must share a common notion of time. The worst-case guarantee for any time-based QoS is directly affected by the accuracy of the global clock. In case of non-real time parallel algorithms, the performance is severely affected by the scheduling strategy used at each node if the applications involve significant communication. Local scheduling that is not based on a global

* This work was funded in part by DARPA, US Navy TASP, and the National Science Foundation

[#] Corresponding Author. Phone: 662 325 8435. Fax: 662 325 8997.

scheme results in low CPU utilization and excessive communication and context switching overheads. Gang scheduling and co-scheduling alleviate this problem by providing simultaneous access to all cluster resources for each job [10]. A fine-synchronized global clock enables the system to implement global schedules with no extra overhead for context switching entire applications concurrently improving their scalability and responsiveness [11].

This paper addresses all of the above requirements, and presents an architecture and implementation of a Myrinet-connected [8] cluster of workstations designed to support high performance distributed real-time applications. The cluster is based on a commodity operating system (Linux) with real time enhancements (Turtle) [1]. The system features a novel Myrinet driver that is adapted to provide predictable end-to-end communication and a high-accuracy, low-overhead global clock.

The rest of this section describes past work in related areas and presents our approach to solving the problems of synchronization, real-time scheduling and communication on Myrinet. Section 2 briefly describes the clock synchronization technique used to achieve high-accuracy. Section 3 discusses the integration of the global clock with the individual CPU schedulers. Section 4 describes the design and implementation of a real-time messaging layer, BDM-RT. Section 5 presents experiments and results that demonstrate a high degree of predictability in message passing and tight synchronization among cluster nodes. Section 6 offers conclusion and presents a synopsis of our future goals.

1.1 Related work in cluster computing

Myrinet [8] has traditionally been used for high-performance computing, with little or no emphasis to providing QoS and/or real-time guarantees. The primary challenge associated with using Myrinet for real-time communication is to ensure bounded fall-through times across the worm-hole switches. Myrinet worm-hole switches have predictably low latencies, that are less than a micro-second only in the absence of switch contention. FM-QoS [17] is a messaging layer that overcomes switch contention by building a global view of time among LANai interfaces [8] in the Myrinet network. FM-QoS uses feedback from the actual latency incurred by specially introduced traffic, to correct for drifts among Real Time Clocks (RTCs) on the individual LANai interfaces. The resulting synchronization enables conflict-free scheduling of link traffic with predictably low latencies between LANai interfaces.

However, resolving switch contention alone is insufficient for ensuring end-to-end latency predictability of Myrinet messages. Contention of shared resources within the sending and receiving nodes is essential. These shared resources comprise the host CPU, the LANai processor, the inter-connecting bus (i.e. PCI bus), limited network buffer space, and the on-board DMA engines [8].

A model for end-to-end periodic real-time communication on Myrinet has been reported by Zhang et al. [19]. A two-level scheduling scheme creates a virtual, slower network for each real-time application across the network. This model is particularly suited for open systems because it does not require intimate knowledge of the timing characteristics of the system, or analysis of global schedulability. The primary advantages of this model are its high resource utilization and simple admission criterion. Implementation of this model was reported as on-going work, citing the implementation challenges involved [19].

1.2 Our approach

Our approach into providing a distributed real-time cluster is largely dictated by the requirements of MPI/RT [6] real-time channels. In particular, our system permits efficient layering of MPI/RT time-based channels with full QoS capability. A channel in MPI/RT is a persistent logical path for unidirectional message transfer between two endpoints (processes). MPI/RT enforces early-binding semantics, wherein an application specifies desired QoS and system resource requirements before entering the real-time mode. MPI/RT can provide three types of "sidedness", that refers to the handshaking exchange between communicating tasks.

Standard asynchronous communication requires two-sided handshaking. A push or pull model illustrates one-sided communication. With pre-established early-binding of channels, it is possible for an application to pre-specify a time or event for sending data on a channel. In this case, it is possible for the middleware to implement data transfers without any handshaking nor explicit user level call. Such a message transfer is termed as zero-sided. We implement such channels as real-time send and receive tasks with time-based QoS meeting the channel's requirements.

The scope of our system however extends beyond MPI/RT and is geared to facilitate high-performance real-time distributed computing by providing: (a) Fine-grain global co-scheduling of distributed tasks to reduce synchronization overheads, (b) end-to-end predictable real-time communication, and (c) guaranteed CPU and network bandwidth.

The design and components of our system are described in detail in the following sections. We implement a high-accuracy fine-grain global clock based on a master-slave scheme to synchronize host clocks. The global clock is integrated with the individual CPU schedulers on each node. On each node, the Turtle scheduler [1] provides bounded-jitter time-based scheduling of computing tasks and communicating "channel" tasks. Communicating tasks shepherd their real-time messages between the host and network interface in their allotted CPU time, eliminating priority inversion. A real-time messaging layer, BDM-RT, guarantees bounded-time protocol processing. Conflict-free global schedules can ensure deterministic end-to-end latencies across the Myrinet network, irrespective of interference from non-real-time Ethernet traffic.

The global clock and scheduler in Turtle are implemented as incremental Linux kernel modules. At initialization, the global clock module is installed and establishes a synchronized time frame. Once the notion of a global time has been established, the Turtle scheduler is installed. The time-based Turtle scheduler uses global time for all dispatch decisions. The following sections describe the implementation of each of these modules.

2 Global Clock on Myrinet

This section summarizes the goals, design and results achieved for the fine-grain global clock implemented using Myrinet. The global clock design and implementation has been described in detail in another paper [2].

2.1 Clock synchronization in high performance real-time clusters

Scalability and low resource overhead are the main challenges for software-based clock synchronization schemes on high performance clusters. High accuracy and predictable resource consumption patterns by the synchronization algorithm are also highly desirable. While high-accuracy permits efficient co-scheduling, predictability of the synchronization algorithm minimizes contention for network resources between real-time applications and the clock module.

Algorithms that solve the Byzantine general's problem [12] require communication that grows as $\theta(n^2)$ with respect to the number of clocks involved in the synchronization. Hence, these algorithms do not scale well with network size due to unacceptably high communication overhead, particularly for large clusters. Hardware based clock synchronization using GPS systems [13] is a viable alternative, but is usually more expensive than a software solution that uses available network.

2.2 Clock synchronization module

We use a master-slave mechanism to achieve internal clock synchronization [14]. This mechanism scales well with network size, requiring $\theta(n)$ message transmissions at every re-synchronization step. The master node periodically broadcasts a 64-bit nanosecond-precision host clock value, using which each slave node builds a synchronized virtual clock. The design uses a novel technique to greatly improve the achievable accuracy, compared to traditional master-slave schemes [2]. Clock values sent by the master are kept "ticking" by adding the protocol-processing delays encountered by clock messages. The transmission latency between the master and slave

LANai interfaces is estimated using network feedback at the master node. The above-mentioned techniques greatly improve the accuracy of the master message as read by the slave, enabling improved clock accuracy. The clock module is scheduled as a periodic real-time task at both the master and slave nodes. Clock messages are transmitted using BDM-RT and consequently incur deterministic latencies and exhibit predictable utilization of the PCI bus, LANai processing time and network bandwidth.

We achieve an accuracy of $\pm 5\mu\text{sec}$ on an 8-node Myrinet cluster, incurring CPU and network bandwidth overhead as low as 0.05%. The algorithm is tolerant to faulty transmission of clock values and occasionally missed clock messages. The next section describes how the global clock is coupled with the scheduler.

3 Integrating global clock with the OS for RT-Tasks

3.1 Turtle Scheduler

Each node in the cluster runs our time-based scheduling variant of RT-Linux v.0.5 [4] called Turtle[1]. Turtle uses RT-Linux's capability to intercept all hardware interrupts, and deliver them to the Linux kernel only when it is on CPU. The Turtle scheduler incorporates a novel strategy based on the Rate controlled scheduling model by Yau and Lam [7]. The scheduling strategy provides the same type of temporal isolation for all real-time tasks as the RT-Mach resource reservation model [5]. The scheduler is designed to allow concurrent execution of real-time tasks, and non-real-time Linux processes. The Linux kernel itself is guaranteed a minimum QoS to ensure that the workstations allow interactive usage.

All real-time tasks in Turtle are periodic and request QoS in the form of a tuple of start time S , end time E , period P , deadline D , and worst case computation time C . The QoS requirement is a contract between a task and the scheduler. A critical deadline is one by which the scheduler must allocate the task another C units of time. Hence, once a task is given its requested computation time, its critical deadline is updated to the next period. The scheduler prioritizes tasks based on critical deadlines, thus ensuring that a misbehaving task does not affect QoS guarantees for other tasks. Such temporal isolation is critical for real-time scheduling on COTS systems that show non-deterministic behavior. Further, if a task's expected time to finish is within a certain bound of its deadline, it is treated as a non-preemptible task.

Turtle uses the APIC timer on Pentium Pro (and higher) processors for scheduling tasks with nanosecond accuracy. Setting the APIC timer alarm has a low overhead ($\sim 500\text{ns}$). Hence, Turtle uses the timer in a one shot mode, instead of periodic time slices enabling task dispatch with arbitrary granularity. At every scheduling instance, the next alarm is set to the minimum of the wakeup time for the next sleeping task T_{wakeup} , and the end of computation time of the current task T_{finish} .

The Linux kernel executes queued bottom half of device drivers at every timer interrupt (typically every 10ms). If Turtle tries to preempt the kernel before the bottom halves have been handled, it suffers a very large interrupt latency. Further, if a DMA is initiated by a bottom half handler, it competes with the CPU for access to the memory bus. In order to avoid these problems, Turtle runs the Linux kernel as a real-time task with a guaranteed QoS of 1ms every 10ms and emulates the timer interrupt for the Linux kernel only when the duration up to the next alarm is sufficiently large. Although created as a hard real-time task, the Linux kernel is treated as a special greedy task that is given extra computation time to pick up any extra cycles as long as there are no other ready real-time tasks.

3.2 Integrating Turtle with the global clock

The Global clock initialization routine also uses the APIC timer to periodically fire a handler that receives clock synchronization messages from the master. The Turtle module is installed after the global clock has been initialized and all nodes synchronized. During

initialization, Turtle obtains the period, computation time and expected arrival time of the next synchronization message from the global clock handler. This information is used to create a real-time task that periodically sends (in case of master) or receives (in case of slaves) synchronization messages. Turtle then creates a real-time task for the Linux kernel itself. The scheduler then registers itself as the APIC timer handler and the system is ready to instantiate other real-time tasks.

4 Real-time messaging

4.1 Messaging architecture

In a distributed real-time system, the communication subsystem is a critical infrastructure [14]. In our architecture, Myrinet is used by MPI/RT channel tasks for real-time messaging, while Ethernet is used by non-real-time Linux processes. The Myrinet network chiefly consists of low-latency worm-hole switches, high-speed Myrinet cables, and programmable Myrinet interface cards. The Myrinet interface [8] consists of the LANai processor, on-board SRAM, two DMA engines for network DMA, and one DMA engine for transferring data between host memory and LANai memory.

BDM-RT is the low-level real-time messaging layer that facilitates efficient layering of the MPI/RT channel abstraction. BDM-RT has been derived from a high-performance messaging layer, BDM [15], and aims at providing deterministic time-based real-time communication with performance as a secondary goal. BDM-RT consists of a host library and a Myrinet Control Program (MCP) that runs on the LANai processor. A complete description of the design of BDM-RT is available in the thesis by Chakravarthi [19]. Non-real-time Linux processes communicate using the Linux TCP/IP stack over the conventional Ethernet network.

4.2 Predictability

Predictability is a primary requirement of real-time messaging layers [16][21][22]. Predictability implies determinism in message latency, protocol processing delays, and access to shared resources involved in communication. Our system uses the layer-by-layer approach [21] to achieve predictability. That is, predictability is built into each software layer, starting from the lowest software layer BDM-RT and working upwards to MIP/RT and the application layer.

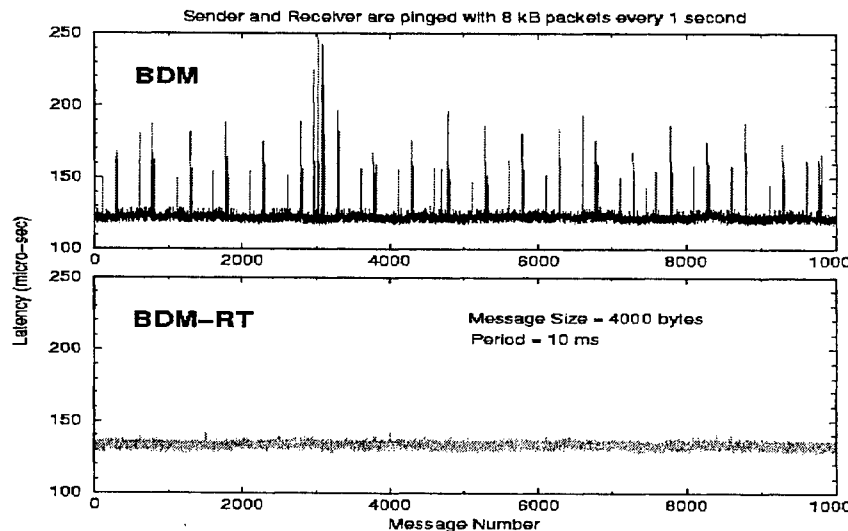


Figure 1 Predictable message passing latency in BDM-RT

4.2.1 Predictable PCI DMA latency

Transfer of data between host memory and LANai memory with predictable latencies is essential to provide predictable end-to-end latency to the layer above BDM-RT. On a COTS system, several devices share the interconnecting bus (e.g. PCI bus) and typically transfer bursty non-real-time data. Bus contention from such devices (e.g., Ethernet interface) causes great unpredictability in the latency of Myrinet PCI DMA transfers.

BDM-RT uses a technique that we have termed “blocking-DMA” to ensure predictable PCI DMA of Myrinet data [1]. BDM-RT blocks the CPU while the Myrinet on-board PCI DMA engines transfer data, thus ensuring that Linux drivers do not initiate a simultaneous PCI transfer. Figure 1 shows the BDM-RT latencies with and without blocking-DMA in the presence of interfering Ethernet traffic from a ping program that sends 8000 bytes of data every second to both nodes.

4.2.2 Bounded-time protocol processing

BDM-RT provides bounded-time response both at the host library and at the MCP. Sending processes should allocate a Host-LANai buffer-pair using a call to `BDMRT_Alloc()` that always succeeds in bounded time. After copying the message to the allocated host buffer, `BDMRT_Send()` transfers the message across the PCI bus in bounded time, which is a function of the message length. At the receiver's node, a call to `BDMRT_Recv()` succeeds and returns in bounded time (a function of message size), provided the message has already been received and stored at the LANai interface. With globally planned schedules and predictable end-to-end latencies, it is safe to assume that messages arrive at the receiver's LANai interface at or before the expected time. It is the responsibility of the layer above BDM-RT to free receive buffers by calling the function `BDMRT_Free()`.

The MCP runs in an infinite loop, and is mainly responsible for initiating and completing DMA transfers to and from the host and the network. For initiating PCI DMA transfers, the MCP responds to flags that are set by the host-library when the user is ready to send or receive data. Network DMA transfers are initiated by the MCP after examining a LANai register and detecting a message (in case of received messages) or upon completion of Host-to-LANai PCI DMA (in case of outward bound messages). All actions of the MCP are non-blocking in nature to ensure bounded duration of the MCP infinite-loop.

4.3 Resource management

The resources that primarily require management to facilitate conflict-free network accesses are the LANai processor, LANai buffer space, PCI DMA bandwidth, and link bandwidth. BDM-RT has been tightly coupled with the Turtle scheduler to manage shared resources without introducing priority inversion. For instance, PCI bus bandwidth is managed by absorbing PCI DMA latency into the computation time parameter (C) of the communicating channel task. This is because BDM-RT functions that trigger PCI DMA transfers do not give up the CPU until the transfer completes. Although this reduces performance, it ensures that no other device driver competes for the PCI bus when an application wishes to transfer data. Link bandwidth and limited receive-buffer space at the LANai are managed by generating conflict-free global schedules based on the global clock. It is important to notice that the high-accuracy global clock, the low-jitter Turtle scheduler, and bounded-time protocol processing by BDM-RT are all required to ensure conflict-free link schedules without requiring a second-level of link scheduling at the MCP. In other words, contention of link bandwidth and Myrinet worm-hole switches is eliminated without explicit synchronization among the MCPs. In this sense, our approach is orthogonal to that of FM-QoS [17] which aims to achieve LANai-level synchronization.

4.4 Priority Inversion

Priority inversion is minimized by migrating most of the protocol processing activity into the tasks' context. Sending tasks marshal their messages to LANai memory, and receiving tasks transfer messages from LANai memory to the host in their allotted CPU slot. This minimizes priority inversion that can be caused at the receiver's LANai interface when a high-priority (early-deadline) message is blocked by another low-priority (late-deadline) message that is currently using the PCI bus. The above-mentioned situation is eliminated by performing LANai-to-host PCI transfers only during the receive task's allotted CPU time.

4.5 QoS sensitivity

QoS-sensitive protocol processing is necessary to meet QoS requirements of real-time tasks [18]. Generally, link traffic is prioritized based on a scheme such as Earliest Deadline First, or Weighted Fair Queuing to ensure that processes with the most stringent QoS requirements are serviced first. On our system, prioritizing is done by the Turtle scheduler based on message deadlines of the communicating tasks (channel tasks). The channel task parameters such as start-time (S), deadline (D), end-time (E) and period (P) are decided by the MPI/RT middle-ware, based on the channel's QoS attributes. The CPU time (C) is set to the worst-case estimate of PCI DMA latency for the channel's message size. The BDM-RT MCP simply dispatches messages handed to it by the host library in FIFO fashion, because messages are already in correct order. At the receiver node, messages are de-multiplexed among the various channel tasks based on message tags.

MPI/RT has to ensure that Myrinet switch contention is absent when global schedules are generated at the "commit" phase [6]. For any given schedule, it is possible to estimate the arrival time of messages at a Myrinet switch because messages incur bounded protocol processing delays before arriving at a switch. Thus it is possible to identify and discard global schedules that can potentially cause switch contention. The sample application in the next section establishes these bounds on messaging latency.

5 Sample Application (Ring)

The following experiments have been performed on an eight node cluster connected by Myrinet. Each node consists of a Pentium Pro 200 MHz processor with Intel FX440 motherboard. A single 32 bit PCI bus (33 MHz) is shared by Ethernet, Myrinet and SCSI interfaces. The Myrinet PCI interface consists of the LANai 4.x processor with 1MB of SRAM. The base operating system used is Linux 2.0.30. Turtle is an enhancement variant of RT-Linux 0.5. The Myrinet network is driven by the BDM-RT messaging library. This configuration represents a typical high performance cluster composed of commercial-off-the-shelf (COTS) desktops. All the results presented here are expected to scale to all newer Pentium and Myrinet architectures.

As previously mentioned zero-sided MPI/RT channels can be implemented using RT tasks that send and receive messages with QoS of the channel. The messaging latency for an application that uses zero-sided communication is an interval from the time at which the sender task is invoked on node i ($T_{\text{send}}(i)$), up to the deadline of the receiver task on node $(i+1)$ ($D_{\text{recv}}(i+1)$). Deadlines for all tasks are set as the sum of their worst case computation time and the scheduling overhead.

The sequence of actions during a message transfer is depicted in Figure 2. To establish accurate bounds on the message passing latency, a sample application was developed that sends messages through the cluster in a ring topology.

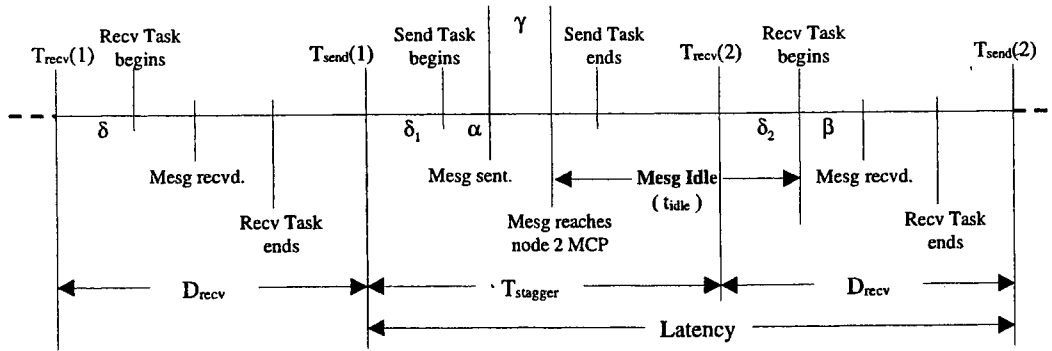


Figure 2: Time-line in messaging

Periodic Sender/Receiver task pairs are initialized on each node. The tasks have starting times of T_{send} and T_{recv} and deadlines of D_{send} and D_{recv} respectively. For any node i , the starting time $T_{recv}(i)$ of the receiver task is

$$T_{recv}(i) = T_{send}(i-1) + T_{stagger} \quad (1)$$

The starting time T_{send} for the sender task is

$$T_{send}(i) = T_{recv}(i) + D_{recv}(i) \quad (2)$$

On getting a message, the receiver task puts it in a shared memory area, which is then forwarded to the next node by the sender task. The send and receive calls are non-blocking and the application fails if a message is not ready when the call is made. The goal of this experiment is to minimize the time the message is idle on the MCP (t_{idle}) as shown in the figure, by reducing $T_{stagger}$.

As shown in the figure, the message reaches remote MCP at $(T_{send}(1) + \delta_1 + \alpha + \gamma)$, where δ_1 is scheduling overhead for node 1, α is the computation overhead of the send call, and γ is actual network latency. Receive task starts retrieving the message at $(T_{recv}(2) + \delta_2)$. Hence, message idle time is minimized if

$$t_{idle} = T_{recv}(2) - T_{send}(1) + (\delta_2 - \delta_1) - (\alpha + \gamma) = 0 \quad (3)$$

Note that T_{send} and T_{recv} are measured with respect to the global clock on nodes 1 and 2. This is subject to a worst case clock variation of $\epsilon = \pm 5\mu s$. Taking that into consideration along with equation (1), the above equation can be recast as :

$$T_{stagger} + \epsilon + (\delta_2 - \delta_1) - (\alpha + \gamma) = 0 \quad (4)$$

Hence,

$$T_{stagger} \equiv \epsilon + (\delta_1 - \delta_2) + \alpha + \gamma \quad (5)$$

To compute the minimum achievable $T_{stagger}$, several tests were done to establish bounds on the actual network latency γ [$1\mu s.. 4\mu s$] and scheduling overhead δ [$4\mu s .. 25\mu s$]. Uncertainty in scheduling overhead arises from hardware/software interrupt latency and cache effects. α is almost a constant at $5\mu s$ since BDM-RT provides deterministic send and receive calls. As a result in the worst case,

$$T_{stagger} \equiv 10 + 21 + 5 + 4 = 40\mu s \quad (6)$$

$T_{stagger}$ was fixed at $45\mu s$ and the experiment was successfully run a number of times with periods of 5ms and 10ms for 10,000 messages. Figure 3 shows the message idle time recorded on the Myrinet board. The minimum idle time is $13\mu s$, which suggests that we may be able to reduce

T_{stagger} to about $35\mu\text{s}$. However, experiments for stagger values under $45\mu\text{s}$ did not always succeed.

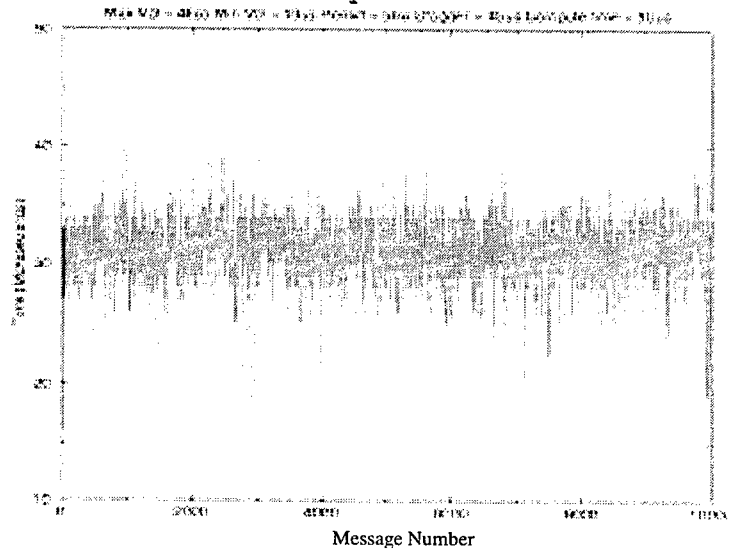


Figure 3 Jitter in waiting time at Lanai

All 8 nodes were pinged every second with 8kB packets on ethernet for the duration of the experiment. Figure 4 shows the actual time when a message is put in the user's buffer with respect to start of the receive task (T_{recv}). The variation in the receive time is about $22\mu\text{s}$, which reflects the jitter in scheduling overhead. The small $4\mu\text{s}$ peaks seen in the graph are introduced by the default RT interrupt handler, that records the occurrence of ethernet interrupt to be serviced when Linux gets back on CPU. The worst case computation time for the receiving task was established at $50\mu\text{s}$. The deadline D_{recv} is set as $75\mu\text{s}$ (worst case computation time + scheduling overhead). As a result, the end-to-end message passing latency including the scheduling overhead is:

$$\text{Messaging Latency} = (T_{\text{stagger}} + D_{\text{recv}}) = 75 + 40 = 115\mu\text{s}. \quad (7)$$

The same experiment was repeated without ethernet traffic, and the jitter remained of the same order, establishing predictability of BDM-RT in face of other PCI traffic.

An MPI/RT library implementation on this platform can use this form of tuning to accurately schedule real-time channels that use zero sided communication.

6 Conclusions

This paper describes the architecture and implementation of a Myrinet cluster of workstations that is tailored for MPI/RT based distributed real-time applications and high performance parallel applications using coscheduling and gang scheduling. The system features a high-accuracy global clock that facilitates time-based coscheduling of real-time tasks using Turtle, while allowing interactive usage of all workstations using Linux. BDM-RT is a low-jitter and deterministic-overhead messaging layer. A sample ring application demonstrates the use of this system and establishes bounds on worst case message passing latency.

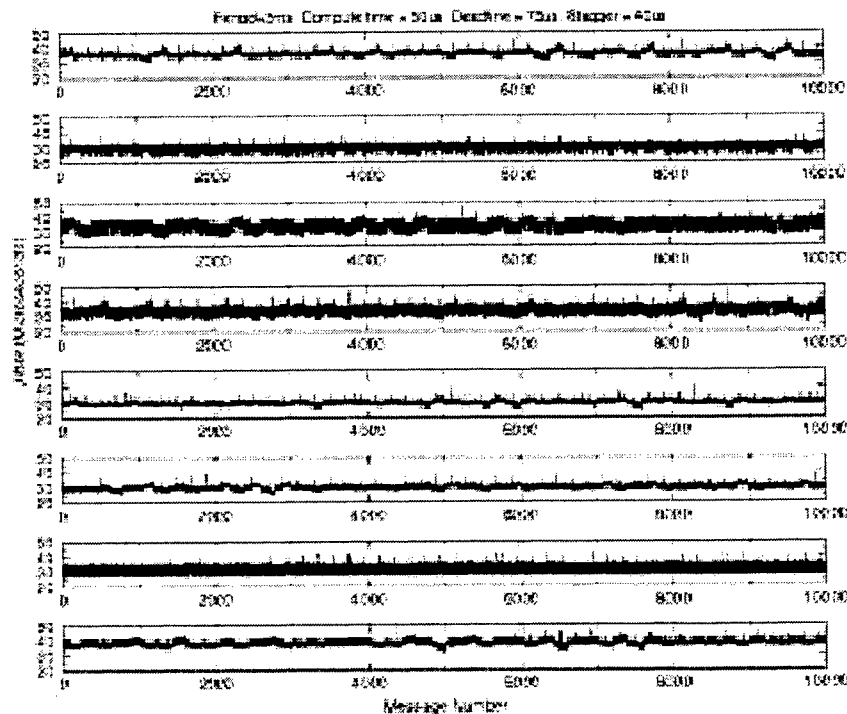


Figure 4 Jitter in message receive time.

7 Future Work

Efforts are underway to implement a synchronized cluster using similar algorithms on GPS hardware. Admission control and generation of global schedules is an area of intense research, and several approaches based on genetic algorithms and reinforcement learning are currently being investigated. An automated global scheduling algorithm will truly exploit the architecture presented here for high performance real-time and non real-time distributed applications. The cluster will serve as a base for the prototype implementation of time-based MPI/RT channels with full QoS support.

8 References

- [1] M. Apte, S. Chakravarthi, A. Pillai, A. Skjellum, and X. Zan. Time based Linux for Real-Time NOWs and MPI-RT. In *Proceedings of the IEEE Real-Time Systems Symposium*, pp 221-222, IEEE Computer Society, Dec 1999.
- [2] S. Chakravarthi, A. Pillai, J. Padmanabhan, M. Apte, and A Skjellum. A Fine-Grain Clock Synchronization Mechanism for QoS Based Communication on Myrinet. Submitted to *International Conference on Distributed Computing 2001*.
- [3] R. Buyya. *High Performance Cluster Computing. Volume 1. Architectures and Systems*. Prentice Hall PTR, New Jersey, NJ. 1999.
- [4] M. Barbanov and V. Yodaikin, Real-Time Linux, *Linux Journal*, March 1996.
- [5] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QoS in real-time Mach, In *Proceedings of Multimedia Japan 96*, April 1996.

- [6] MPI/RT: Real-Time Message Passing Interface Standard 1.0. <<http://www.mpirt.org>>.
- [7] David K. Y. Yau and Simon S. Lam. Adaptive Rate-Controlled Scheduling for Multimedia Applications. In *Proceedings of ACM Multimedia Conference*, Boston MA, November 1996.
- [8] Myricom Inc. LANai3/4 Documentation, 1997.
<<http://www.myri.com/scs/L3/documentation.html>>
- [9] F.A.B. Silva and I.D. Scherson. Concurrent Gang: Towards a Flexible and Scalable Gang Scheduler. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, Natal, Brazil, September 1999
- [10] B. Abali, C. Stunkel, and C. Benveniste. Clock Synchronization on a Multicomputer. *Journal of Parallel and Distributed Computing*, 40(1) 1997, 118-130.
- [11] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of Gang-Scheduling on Workstation Cluster. In *Proceedings of IPPS'96 Workshop*, pp.76-83. April 16, 1996
- [12] L.Lamport, R. Shostak and M. Pease. The Byzantine Generals problem. *ACM Transactions on Prog. Lang. System* 4,3, pp 382-401, July 1982
- [13] B. Sterzbach. GPS-based Clock Synchronisation in a Mobile, Distributed Real-Time System. *Real-Time Systems*, Vol.12, No. 1, 1997.
- [14] J.A. Stankovic, and K. Ramamritham. Clock Synchronization. *Advances in real-time systems*, pp 503, 1993
- [15] Henley, G., N. Doss, T. McMahon, and A. Skjellum. BDM: A multiprotocol Myrinet Control Program and host application programmer interface. Technical Report. MSU-EIRS-ERC-97-3, Mississippi State University, 1997.
- [16] Cilingiroglu, A., S. Lee, and A. Agrawala. Real-time communication. Technical Report. CS-RT-3740, University of Maryland, College Park, 1997.
- [17] Connelly, K. and A. Chien. FM-QoS: Real-time communication using self-synchronizing schedules. In *Proceedings of Supercomputing 1998*.
- [18] Mehra, A., A. Indiresan, and K. G. Shin. Structuring communication software for Quality of Service guarantees. In *Proceedings of the Real-Time Systems Symposium*, pp 144-154. 1996.
- [19] Charkravarthi, S. Predictability and performance factors influencing the design of high-performance messaging layers. Masters Thesis. Dept. of Computer Science, Mississippi State University. 2000.
- [20] Zhang L. Y., J. W. S. Liu, Z. Deng, and I. Philip. Hierarchical scheduling of periodic messages in open system. In *Proceedings of the Real-Time Systems Symposium*, pp 350-359. 1999
- [21] J. Stankovic and K. Ramamritham. What is predictability for real-time systems?. *Real-Time Systems*. Vol 2, pp. 247-254. 1996
- [22] C. Lee, K. Yoshida, C. Mercer and R. Rajkumar. Predictable communication protocol processing in Real-Time Mach. In *Proceedings of 2nd Real-Time Technology and Applications Symposium*. 1996.

PREDICTABILITY AND PERFORMANCE FACTORS INFLUENCING THE
DESIGN OF REAL-TIME MESSAGING LAYERS

By

Srigurunath Chakravarthi

Copyright by

Srigurunath Chakravarthi

2000

A Thesis

Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

May 2000

PREDICTABILITY AND PERFORMANCE FACTORS INFLUENCING THE
DESIGN OF REAL-TIME MESSAGING LAYERS

By

Srigurunath Chakravarthi

Approved:

Anthony Skjellum
Associate Professor of
Computer Science
(Director of Thesis)

Donna Reese
Associate Professor of
Computer Science
(Committee Member)

Daniel Linder
Assistant Professor of
Electrical and Computer Engineering
(Committee Member)

Rainey Little
Associate Professor of
Computer Science and
Graduate Coordinator

A. Wayne Bennett
Dean of the College of Engineering

Name: Srigurunath Chakravarthi

Date of Degree: May 13, 2000

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: PREDICTABILITY AND PERFORMANCE FACTORS
INFLUENCING THE DESIGN OF REAL-TIME MESSAGING
LAYERS

Pages in Study: 92

Candidate for Degree of Master of Science

Several high-performance communication layers such as BDM, GM, and FM have been implemented over the high-speed Myrinet network. Their design and interface are generally inadequate for both applications to specify QoS requirements, and for the system to perform QoS-sensitive protocol processing and resource management. The primary contribution of this thesis is the design and implementation of BDM-RT, the first known hard real-time Myrinet messaging layer. BDM-RT can provide end-to-end guarantees on message latency and bandwidth based on a fine grain global clock and bounded response time at the network interface.

This thesis works to prove the following hypothesis: There is a fundamental dichotomy between the design of low-level real-time and high-performance messaging layers. This implies that performance goals of high-speed messaging layers influence their predictability, and that predictable communication requires performance trade-offs. The hypothesis is proved by analysis of design of BDM and BDM-RT and by experimental verification.

DEDICATION

I would like to dedicate this to my first teachers — my mother and father.

ACKNOWLEDGMENTS

Words cannot express my gratitude to my advisor Dr. Anthony Skjellum for giving me a chance to work under his guidance. He has been a constant source of motivation through various stages of this work. My sincere thanks to Dr. Donna Reese and Dr. Daniel Linder who have motivated me both as excellent teachers and as committee members. I would like to thank co-researchers Greg Henley, Manoj Apte, and Anand Pillai without whose involvement this work would not have been possible. Thanks to Angadi Raghavendra for his help with editing in \LaTeX . Finally, I would like to thank all my teachers, specially Dr. Rayford Vaughn, for making my masters program a truly enriching and memorable experience.

TABLE OF CONTENTS

DEDICATION	
ACKNOWLEDGEMENTS	
LIST OF TABLES	
LIST OF FIGURES	
CHAPTER	
I. INTRODUCTION	
1.1 Background	
1.2 Motivation	
1.3 Hypothesis	
1.4 Basis	
1.5 Scope	
1.6 Contributions	
1.7 Organization of this thesis	
II. SYSTEM ARCHITECTURE	
2.1 Hardware architecture	
2.1.1 Myrinet	
2.2 Software architecture	
2.2.1 PromisQoS framework	
2.2.2 Clock synchronization module	
III. REAL-TIME COMMUNICATION	
3.1 Predictability	
3.2 Resource management	
3.3 QoS sensitivity	
3.4 Provision for best-effort traffic	
3.5 Traffic isolation	

CHAPTER

3.6	Time-bound protocol processing	
3.7	Real-time messaging on Myrinet	
3.8	Summary	
IV.	HIGH-PERFORMANCE MESSAGING	
4.1	Design issues	
4.1.1	Host memory to network interface transfers	
4.1.2	DMA vs. PIO	
4.1.3	DMA initiation and completion	
4.1.4	Message receipt mechanism	
4.1.5	Buffer queue management	
4.1.6	Zero-copy mechanism	
4.1.7	Reliability	
4.2	Timeliness and resource management	
4.3	Summary	
V.	DESIGN OF BDM AND BDM-RT	
5.1	Messaging Protocol	
5.2	BDM design	
5.2.1	BDM queues	
5.2.2	Message flow	
5.2.3	Message receipt	
5.2.3.1	Polling vs. interrupt-based	
5.2.3.2	The receive function	
5.2.4	Host-LANai data transfer	
5.2.5	Message receipt order	
5.3	BDM-RT design	
5.3.1	BDM-RT queues	
5.3.2	Message flow	
5.3.3	Message receipt	
5.3.4	Host-LANai data transfer	
5.3.5	Resource management	
5.3.5.1	Network bandwidth	
5.3.5.2	PCI bandwidth	
5.3.5.3	LANai buffer space	
5.3.5.4	LANai CPU time	
5.3.6	Priority inversion	
5.3.7	Comparison with FM-QoS	
5.4	Summary	

CHAPTER

VI. COMPARISON OF BDM AND BDM-RT

6.1	Performance-predictability trade-offs
6.2	BDM and BDM-RT: Design differences
6.2.1	MCP main-loop
6.2.1.1	BDM
6.2.1.2	BDM-RT
6.2.2	Message receipt
6.2.2.1	BDM
6.2.2.2	BDM-RT
6.2.3	Sender-side host to LANai transfer
6.2.3.1	BDM
6.2.3.2	BDM-RT
6.2.4	Header information
6.2.4.1	BDM
6.2.4.2	BDM-RT
6.2.5	Sender-Side message delivery order
6.2.5.1	BDM
6.2.5.2	BDM-RT
6.3	Summary

VII. EXPERIMENTS, RESULTS, AND ANALYSIS

7.1	Experiment setup
7.1.1	Hardware and software
7.1.2	Timers used
7.1.3	LANai storage limitation
7.1.4	Metric overhead
7.1.5	Accuracy and error bounds
7.2	Latency measurements
7.2.1	Expected results
7.2.2	Actual results
7.3	Bandwidth measurements
7.3.1	Expected results
7.3.2	Actual results
7.4	Latency jitter
7.4.1	Expected results
7.4.2	Actual results
7.5	Effect of external traffic
7.5.1	Expected results
7.5.2	Actual results
7.6	Summary of results

CHAPTER	
VIII. CONCLUSIONS	
8.1 Future work	
REFERENCES	
APPENDIX	
A. HARDWARE MECHANISMS FOR PREDICTABILITY	
A.1 PCI register initialization	
A.2 Customizing PCI arbitration	
A.3 Dual PCI bus	
B. BDM AND BDM-RT API	
B.1 BDM API Functions	
B.2 BDM-RT API Functions	

LIST OF TABLES

TABLE

5.1	BDM Queues
5.2	BDM-RT Queues and Lists

LIST OF FIGURES

FIGURE

2.1	Topology of a Myrinet Network
2.2	LANai Interface
2.3	Software Architecture
2.4	Software Architecture of PromisQoS
2.5	Global Clock
4.1	Raw SBus Bandwidth comparison on UltraSparc running Solaris . .
4.2	Raw PCI Bandwidth comparison on Intel x86 running Linux
5.1	BDM Send Queues
5.2	BDM Receive Queues
5.3	Receive Semantics for BDM
5.4	Send Semantics for BDM
5.5	BDM-RT Receive Queues and Lists
5.6	Receive Semantics for BDM-RT
5.7	Send Semantics for BDM-RT
6.1	Comparison of Receive Semantics between BDM and BDM-RT . . .
6.2	Comparison of Send Semantics between BDM and BDM-RT
6.3	BDM Send Semantics: Contention with Linux for PCI bus
7.1	Latency for short messages
7.2	Latency for long messages
7.3	Bandwidth

7.4	BDM: Latency Jitter
7.5	BDM-RT: Latency Jitter
7.6	Effect of non-real-time ethernet traffic

CHAPTER I

INTRODUCTION

1.1 Background

The Real-Time Messaging Passing Interface (MPI/RT) (Kanevsky, Skjellum, and Watts 1997) is a recently emerging middleware standard for distributed real-time computing. MPI/RT provides a notion of Quality of Service (QoS) for development of real-time applications. MPI/RT supports three real-time programming paradigms, namely — Time-based, Priority-based, and Event-driven programming models. In order to be able to provide QoS guarantees for applications, the MPI/RT middleware requires real-time support from the Operating System and hardware that lie below it. In this regard, MPI/RT is said to impose “middle-down” requirements, that is, potential modification of operating systems in order to meet its goal. No known implementation of MPI/RT with full QoS guarantees exists to date. This is owed to both the relative infancy of the standard and the challenges involved in its implementation. MPI/RT requires support from a distributed real-time operating system that provides real-time services that map well to the MPI/RT QoS model. PromisQoS (Apte et al. 1999, 221) is an envisioned time-based real-time operating system that is being designed and implemented currently at Mississippi State University. The primary goal of PromisQoS is to enable a prototype implementation of MPI/RT that provides full QoS support for time-based real-time computation and communication. The challenges involved in the development of PromisQoS are many. The envisioned

system is required by MPI/RT primarily to provide bounded-delay messaging, bounded-jitter process scheduling on a global clock, and temporal isolation of processes to avoid contention of shared resources. The system is also required to offer high performance by providing high bandwidth, low latency, and low overhead communication. The basic building blocks of PromisQoS are its EDF TURTLE scheduler, a high-precision global clock, and a real-time messaging layer BDM-RT. These components are those most essential for the implementation of the MPI/RT Time-based real-time Channel abstraction (Kanevsky, Skjellum, and Watts 1997). This thesis is based on the development of the real-time messaging layer BDM-RT for PromisQoS.

1.2 Motivation

The motivation for this thesis work arises from the requirement of a messaging subsystem with at least the following characteristics: Predictability and QoS support, high bandwidth and low latency communication for high-performance real-time distributed applications, and good mapping between channel abstraction of MPI/RT and messaging system software.

Myrinet is a high speed gigabit per second networking technology that is being used widely for high-performance distributed computing. Initial experiments with Myrinet network latency have demonstrated its suitability for predictable communication as well. Given the dual goals of performance and predictability for MPI/RT applications, Myrinet appears to be a suitable hardware platform for the development of a messaging subsystem that allows efficient layering of MPI/RT.

Several high-bandwidth, low-latency messaging layers have been written over Myrinet. Examples include Fast Messages (FM) (Pakin, Lauria, and Chien 1995)

from University of Illinois, Urbana Champaign, Glenn's Messages (GM) (Myricom, Inc. 1999) from Myricom, Inc., BullDog MCP (BDM) (Henley et al. 1997) from Mississippi State University, LANai Active Messages (LAM) (von Eicken et al. 1992, 256) from University of California, Berkeley, and Basic Interface for Parallelism (BIP) (Prylli 1997) from University of Lyon, France. Most of these messaging layers are used in distributed computing and are typically layered under a middleware such as MPI (Snir et al. 1995) or PVM (Sunderam et al. 1994, 531). However, real-time messaging over Myrinet is not currently popular. FM-QoS (Connelly and Chien 1997) is the only known QoS based messaging layer, but is not fundamentally a hard real-time messaging layer as it cannot provide guarantees on end-to-end latency.

This thesis is motivated by the development of BDM-RT, the first global-clock based real-time messaging layer on the high-speed Myrinet network (Boden et al. 1995, 29). BDM-RT was developed by the author of this thesis by re-engineering the high-performance messaging layer BDM (Henley et al. 1997). The primary goal of BDM-RT is to support the implementation of Time Based MPI/RT Channels with full QoS guarantees. Performance is also important, but is a secondary goal. The goal of providing QoS guarantees influences its design and makes BDM-RT considerably different from traditional high-performance messaging layers. The motivation of this thesis lies in demonstrating the unsuitability of high-performance messaging subsystems for real-time communication, even if performance is an important secondary goal for the real-time system. Further, it constructs an alternative messaging layer to provide predictable communication and analyzes its effect on the performance of the system.

1.3 Hypothesis

This thesis hypothesizes that there is a fundamental dichotomy between the design of low-level real-time and high-performance messaging layers. This work discusses the fundamental design differences between real-time and high-performance messaging subsystems. This thesis works to show that the goals of performance and predictability do not go hand-in-hand and that trade-offs are often essential to improve one at the cost of the other. As a result, messaging systems designed with the primary goal of performance are fundamentally unsuitable for QoS based communication and hence cannot be used for a high-quality implementation of real-time middleware such as MPI/RT.

1.4 Basis

The basis of the stated hypothesis is from initial experimentation on latency jitter¹ of periodic messages using BDM, and the design similarities of BDM with other popular HP messaging layers such as FM and GM.

As pointed out by Mehra, Indiresan, and Shin (1996), a QoS-sensitive messaging layer must be able to provide guaranteed bounds on transfer latencies at the minimum. Without a guaranteed limit, such a layer cannot organize its transfers so as to meet the QoS requirements of the different applications that it services. A real-time messaging layer in general needs to be aware of the current traffic load (Connelly and Chien 1997) in order to plan its transfers in the presence of real-time tasks with differing QoS requirements and best-effort tasks. Alternately, the CPU scheduler should be closely coupled with the messaging subsystem so as to control the network traffic by planning appropriate global CPU

¹Latency jitter is a measure of variation in message latency incurred by individual messages of the same length exchanged between the same end-points.

schedules for all communicating processes across the network. Messaging activity has to be coupled with CPU schedules to synchronize receiving processes with message arrival and to avoid buffer overflows. These basic requirements of a real-time messaging subsystem are absent in messaging systems designed for high-performance alone.

1.5 Scope

This work is based on real-time messaging software designed over Myrinet (Boden et al. 1995, 29). On a general level, this work is particularly applicable to the design of real-time software on architectures with any or all of the following features:

- High speed networks where link bandwidth exceeds host-to-network interface bandwidth
- Networks with programmable network interfaces, and bus master DMA capability
- Cut-through switched networks as opposed to conventional packet-switched networks
- Systems where performance as well as predictability are important goals

Design choices that have been presented and discussed for real-time communication are chiefly applicable to the MPI/RT time-based real-time communication model, although some of these principles apply equally well to systems based on the priority-based and event-based models.

1.6 Contributions

The contributions of this thesis work are as follows:

1. By demonstrating that real-time messaging subsystems impose significantly different design requirements as compared to their high-performance counterparts, this work shows that it is not possible to layer a real-time middleware such as MPI/RT on an existing high-performance messaging layer when QoS guarantees are required.
2. Research in real-time messaging systems design has been focused around conventional network hardware such as ethernet. The design of BDM-RT introduces new concepts such as “blocking DMA” that address the issue of resource management and traffic isolation in the presence of a network processor and on-board DMA engines.
3. There have been several implementations of the real-time channel paradigm over packet-switched networks (Mehra, Indiresan, and Shin 1995) (Cilingiroglu, Lee, and Agrawala 1996). However, this is by far the first known implementation of a real-time messaging layer that supports direct layering of real-time channels on a worm-hole routed, cut-through switched high-speed network such as Myrinet.
4. BDM-RT is the first known Myrinet messaging layer that implements a fine-grain global clock to facilitate global message schedules.
5. BDM-RT and PromisQoS provide a testbed for a prototype implementation of Time-based MPI/RT Channels with full QoS support. To date, no prototype of MPI/RT with full QoS support has been developed. Successful

implementation of MPI/RT over PromisQoS will serve as a proof of concept for the feasibility of implementation of the MPI/RT Time Based Channel.

6. By demonstrating the dichotomy between the design requirement of high-performance and real-time messaging layers, this thesis paves the way for the design of high-speed network interface hardware that is particularly suitable to both high-performance and real-time distributed computing.

1.7 Organization of this thesis

The remainder of this work is organized as follows. Chapter 2 introduces the hardware and software configuration of the system that BDM and BDM-RT are implemented on. Chapter 3 reviews the basic requirements of real-time communication subsystems based on a literature review of existing systems and past research in real-time communication. Chapter 4 reviews the goals and design choices of high-performance messaging layers on Myrinet. Chapter 5 presents the design of BDM and BDM-RT and discusses the rationale behind their design. Chapter 6 analyzes in detail the unsuitability of BDM for real-time communication as well as the impact on performance of BDM-RT at the cost of improved predictability as compared to BDM. Chapter 7 outlines the experiments that were carried out to support the hypothesis and discusses their expected and actual results. Chapter 8 offers conclusions and suggested future work.

CHAPTER II

SYSTEM ARCHITECTURE

This chapter describes the hardware and software architectures of the distributed system in which BDM and BDM-RT operate. This description has been presented to allow the reader to better understand the influence of the hardware architecture on the design of messaging layers over Myrinet.

2.1 Hardware architecture

The system consists of a cluster of PCs connected by a single 8-port Myrinet switch. The chosen hardware configuration represents a typical cluster of COTS desktops, in keeping with the scope of the hypothesis. Each host consists of a Pentium Pro II 200 MHz processor, and uses the Intel FX440 chip-set. A single 32-bit 33MHz PCI bus is shared by Ethernet, Myrinet and SCSI interfaces. This configuration is representative of recent generation PC architectures. The Myrinet PCI interface consists of the LANai 4.x processor with 1 MB of SRAM. The Myrinet interface is described below.

2.1.1 Myrinet

Myrinet is a widely used high-speed network technology in high-performance domains. Myrinet currently supports full-duplex connections operating at 1.28 Gigabits/sec, and Myrinet cut-through switches route based on message header bits, with latencies as low as 0.5 microseconds. Figure 2.1 shows a Myrinet network

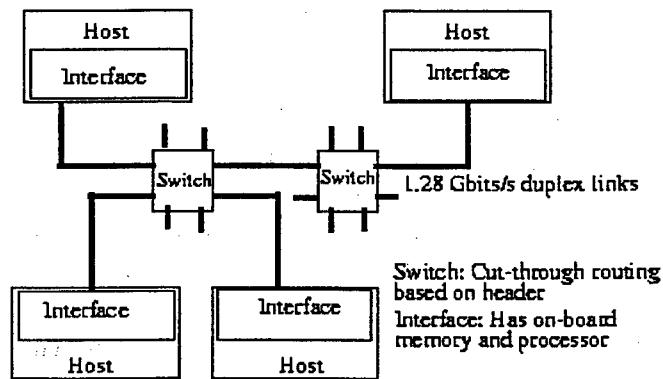


Figure 2.1: Topology of a Myrinet Network

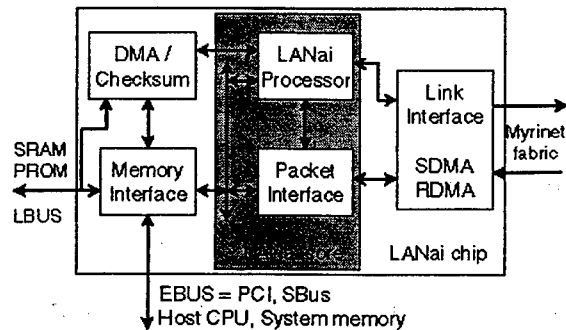


Figure 2.2: LANai Interface

of four hosts connected using two 8-port switches. Myrinet networks provide a degree of reliability in the Link Layer itself, with less than 1 bit-error for every 10^{15} bits transmitted. The recent generation Myrinet network interface typically consists of an on-board processor (the "LANai" processor), about 1 MB of SRAM and three Direct Memory Access (DMA) engines. The interface is shown in figure 2.2.

The three DMA engines are meant for DMA transfers to the network, from the network, and between host and LANai memory. A custom built program called the Myrinet Control Program (MCP) runs on the LANai processor. The LANai

4.x series boards also contain a 32-bit Real Time Clock (RTC) that ticks every 0.5 μ sec.

2.2 Software architecture

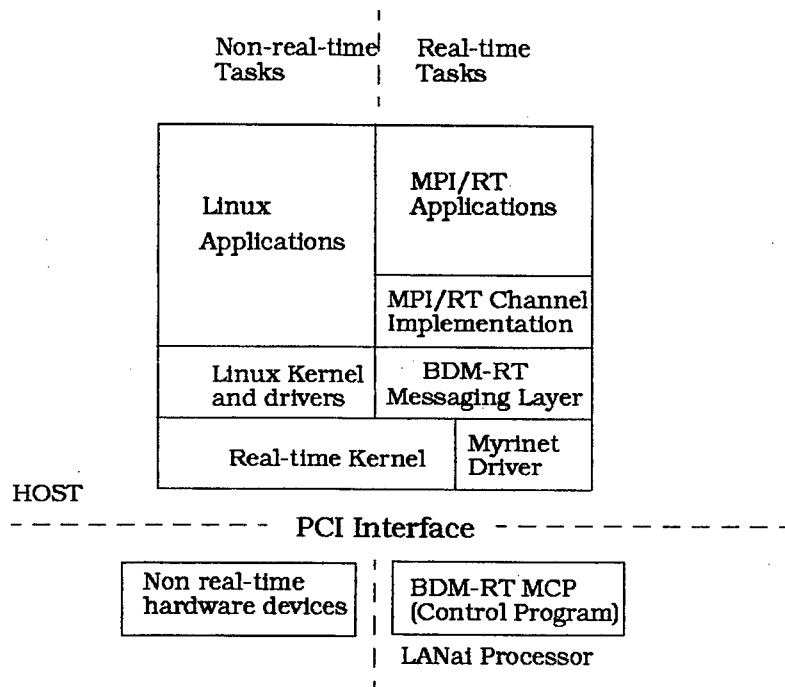


Figure 2.3: Software Architecture

Figure 2.3 offers a broad picture of the various software layers in the system. Real-time and non-real-time software layers are clearly segregated as shown by the vertical dividing line in the figure. Non-real-time applications run as processes on Linux, which itself is serviced by a real-time kernel. Real-time application programs are written over the middleware MPI/RT which is layered on top of the messaging library BDM-RT. At the bottom is a real-time kernel that encapsulates the hardware from the upper layers. For real-time communication, BDM-RT avails itself of services provided by the Myrinet device driver. On the Myrinet

interface, the custom-built Myrinet Control Program (MCP) executes on the on-board LANai processor. The messaging library communicates with the MCP using shared memory flags on the LANai memory.

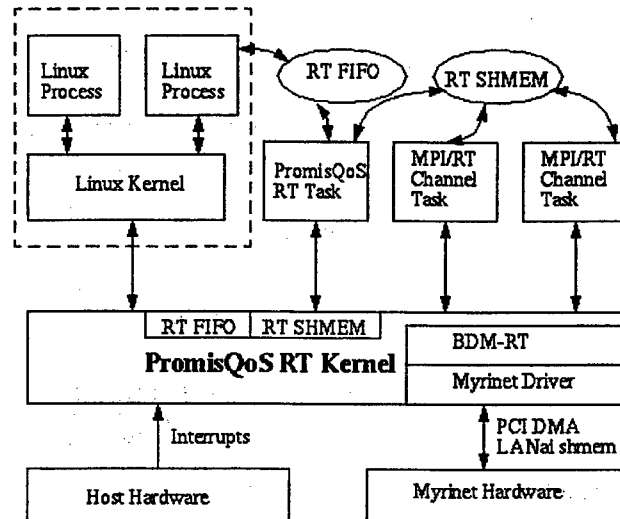


Figure 2.4: Software Architecture of PromisQoS

2.2.1 PromisQoS framework

A detailed view of the PromisQoS Operating System is shown in figure 2.4. The building-blocks of PromisQoS come from the RT-Linux (Barbanov and Yodaiken 1996, 19) framework. Real-time tasks (RT tasks) run in kernel context over the real-time PromisQoS real-time kernel (RT Kernel). Linux itself runs as a special process that is guaranteed at least 1 ms CPU time every 10 ms. The RT kernel buffers all hardware interrupts and delivers them as software interrupts to Linux, when Linux is scheduled on the CPU (Barbanov and Yodaiken 1996, 19). RT tasks in a single node communicate with other RT-tasks and best-effort Linux processes using either PromisQoS's shared memory (RT SHMEM) (Apte et al.

1999, 221) or RT-Linux's RT-FIFOs (Barbanov and Yodaiken 1996, 19). Channel message transfer occurs across the Myrinet network via the BDM-RT library. Each MPI/RT channel is implemented as an real-time task that performs communication using BDM-RT.

PromisQoS uses the TURTLE Earliest-Deadline-First scheduler (Apte et al. 1999, 221), capable of scheduling periodic real-time tasks with periods as low as 60 μ sec. Every real-time task specifies the following five task parameters to the TURTLE scheduler: start-time (S), period (P), deadline (D), computation (C), and end-time (E). All admitted real-time tasks are guaranteed C units of computation time every period, before the deadline D elapses relative to the start of the period.

2.2.2 Clock synchronization module

PromisQoS uses a tightly synchronized global clock to schedule processes across the network. The individual on-chip APIC Time-Stamp Counter on each machine drifts with respect to the others, and varies with changes in the ambient temperature, pressure and other effects. Hence a mechanism is required to keep individual clocks synchronized. Internal clock synchronization (Stankovic and Ramamritham 1993), that is synchronization of clocks with respect to one another but not with respect to the outside world, is achieved by using a master-slave mechanism. One of the nodes plays the master's role and periodically broadcasts the master time-stamp to all slaves across the Myrinet network. Every slave maintains a virtual clock that is updated with the receipt of each master-clock value at every re-synchronization period. The master and slave clocks run as real-time tasks with a period of 1 second.

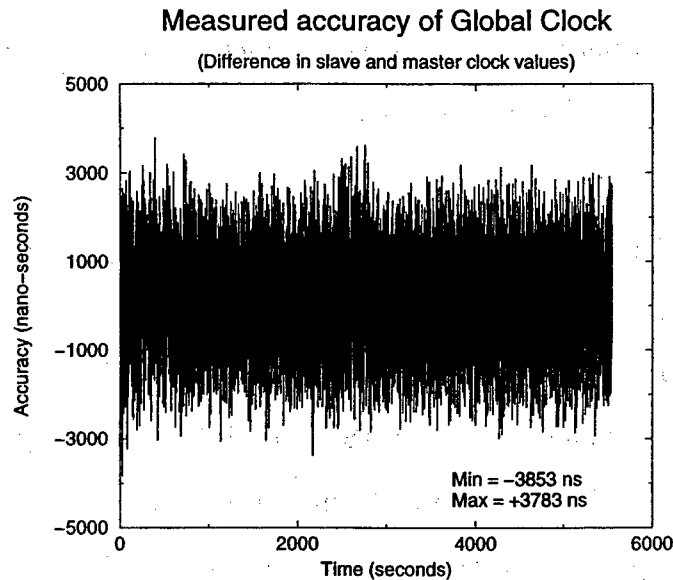


Figure 2.5: Global Clock

Figure 2.5 shows the difference between the slave's virtual clock and the master's clock at every re-synchronization stage. This gives a measure of the accuracy of the global clock. This mechanism achieves synchronization with an accuracy of $\pm 4 \mu\text{sec}$. This is relatively fine grain accuracy for software based clock synchronization. This fine grain accuracy is a result of introducing special mechanisms in BDM-RT that keep the master's clock "ticking" while in transit. It is known that the accuracy of master-slave virtual clocks is limited by the jitter in message transit time. In the case of Myrinet, end-to-end latency jitter far exceeds $4 \mu\text{s}$. BDM-RT reduces the uncertainty in clock message latency by recording message delays incurred by the master clock message in various stages of its transit. Using the recorded delays, slaves can estimate the master clock value at the time of receipt of the clock message with high accuracy. The clock synchronization algorithm is tolerant to faults arising from missed clock

messages. Support for the abovementioned high quality transmission of clock messages was written as a part of BDM-RT development by the author. The fine-grain clock synchronization algorithm and its interface with the rest of the system were developed by collaborators Pillai and Apte (Apte et al. 1999, 221).

CHAPTER III

REAL-TIME COMMUNICATION

In a distributed real-time system, the communication subsystem is critical infrastructure (Stankovic and Ramamritham 1993). The communication subsystem must behave deterministically with respect to communications delays and network resource access times. Some principles of real-time communication based on past research are presented below. These principles govern the design of real-time messaging layers such as BDM-RT. While this chapter presents design essentials for real-time messaging subsystems in general, a detailed discussion of the design of BDM-RT appears in chapter 6.

3.1 Predictability

Predictability is a primary requirement of real-time messaging layers (Cilingiroglu, Lee, and Agrawala 1996) (Stankovic and Ramamritham 1990, 247) (Lee et al. 1996). Predictability implies determinism in message latency, protocol processing delays, and access to shared resources involved in communication including the requested bandwidth. Two popular approaches to achieving predictability in complex real-time systems are the layer-by-layer approach and the top-layer approach (Stankovic and Ramamritham 1990, 247). A real-time system can be broadly viewed to be built of various hardware and software layers such as semiconductor components, the hardware architecture layer, the operating system layer, a middleware layer, and the application layer. The layer-by-layer

approach is based on the assumption that a higher layer is predictable, if and only if, the lower layer is predictable, while the top-layer denies the validity of this assumption. To provide 100% guarantees on deadlines, as is required by critical tasks in a hard real-time system, a layer-by-layer approach is required (Stankovic and Ramamritham 1990, 247). PromisQoS adopts the layer-by-layer approach to enable the layering of a QoS based MPI/RT middleware with predictable messaging support from BDM-RT layer.

3.2 Resource management

A real-time messaging subsystem takes into account all the shared resources used by it and eliminates contention for these resources among the various processes. Resource usage is managed by the subsystem either by providing a resource reservation model as in RT-Mach (Lee et al. 1996), or by arbitrating concurrent accesses based on priority or deadline of the processes involved (Mehra, Indiresan, and Shin 1996). The system should have a mechanism to avoid priority inversion, which can easily interfere with meeting QoS guarantees. The primary resources involved in communication are as follows: network buffer space, host CPU cycles, network co-processor CPU cycles, physical network media, and shared system buses such as the PCI bus (Solari and Willse 1998). In a time-based messaging system, resource contention should ideally be absent or should be resolved in a known, bounded time. Sometimes, the accessibility of one or more system resources may be dependent on the accessibility of another resource. An example of this is that memory, as a resource, can be accessed by a process only when it is scheduled on the CPU, that is, when it has access to the CPU. The RT-Mach team (Lee et al. 1996) has incorporated this into the notion of a

controlling-resource and controlled-resource into their resource reservation model, thus enabling centralized management of multiple related resources. This model is however inapplicable in the presence of bus-master capable devices such as the LANai PCI DMA engine. The PCI bus can be treated as a controlled resource with the CPU as the controlling resource only if all accesses to the PCI bus can originate from processes that are currently using the controlling resource, that is, the CPU. In our case, the MCP can initiate PCI activity independent of the host CPU, thus causing an inconsistency with the abovementioned requirement. Chapter 5 explains how BDM-RT addresses the PCI bus isolation problem.

In cut-through worm-hole routed networks such as Myrinet, contention for the Myrinet switch needs to be specially addressed. Packets arriving at a switch at the same time with the same out-going port can block one another, introducing unpredictability in network transmission latency. Global schedules of network traffic are essential to avoid switch contention. FM-QoS (Connelly and Chien 1997) is a QoS based messaging layer on Myrinet that minimizes the problem of switch contention by synchronizing link traffic based on feedback from network DMA latencies. FM-QoS is described in section 3.7.

3.3 QoS sensitivity

A simple FIFO delivery of messages does not suffice to support the different QoS requirements of real-time tasks, because FIFO delivery amounts to ignoring differences in QoS requirements between processes (Mehra, Indiresan, and Shin 1996). Link traffic should be prioritized based on a scheme such as Earliest Deadline First, or Weighted Fair Queuing to ensure that processes with the most stringent QoS requirements are serviced first (Mehra, Indiresan, and Shin

1996). In our context, MPI/RT provides QoS sensitivity by appropriately assigning channel task parameters such as period, deadline, and computation time during the MPI/RT Commit phase (Kanevsky, Skjellum, and Watts 1997). In other words, channel messages are implicitly prioritized based on the CPU schedules.

3.4 Provision for best-effort traffic

Real-time subsystems are normally designed to support a mixture of best-effort and real-time traffic (RT traffic). In order to ensure that best-effort traffic does not interfere with meeting the QoS requirements of RT traffic, it is required that these forms of traffic be processed separately. The design should also ensure fairness to best-effort traffic by not starving it. The system must clearly separate resources such as buffer space, link bandwidth and CPU time for protocol processing between RT and best-effort traffic. For example, in (Mehra, Indiresan, and Shin 1996) three message queues are used: one for real-time messages that are "on-time" with respect to their deadline (highest priority), one for best effort traffic (medium priority), and the third for real-time messages that are significantly early compared to their deadline (least priority). This avoids starvation of best-effort tasks, while still meeting the QoS requirements of the real-time tasks.

BDM-RT does not have explicit support for best effort traffic because channel priorities are assigned by the MPI/RT Commit function, which is transparent to BDM-RT. Provision for best-effort aperiodic traffic is planned as a future goal because of the complexity involved in admission control of such tasks. In the current framework, BDM-RT can admit best-effort tasks as periodic real-time tasks with relaxed deadlines. This ensures fairness to best-effort tasks and at the same time ensures that real-time tasks meet their deadlines. However, running best-

effort tasks as strictly periodic tasks makes BDM-RT insensitive to the bursty nature of best-effort traffic.

3.5 Traffic isolation

The messaging subsystem must provide traffic isolation by forcing applications to abide by their requested QoS requirements. A misbehaving task should be disallowed from causing a failure to meet other tasks' QoS requirements. In the PromisQoS framework, the MPI/RT middleware and the TURTLE scheduler together ensure that MPI/RT channels adhere to their requested network resource usage. MPI/RT polices PCI bus usage and network bandwidth usage by performing run-time checks on the length of data transferred by each channel, during every period. TURTLE checks bandwidth utilization of channel tasks by policing its allotted CPU time. MPI/RT will produce an error if channel tasks attempt to transfer more data than requested at the MPI/RT Commit phase (Kanevsky, Skjellum, and Watts 1997).

3.6 Time-bound protocol processing

Protocol processing time should be predictable in order to achieve predictable end-to-end latency. Activities such as fragmentation, re-assembly, and queue access routines should be time-bounded. Priority inversion in protocol processing is of specific concern. For example, priority inversion can occur if the protocol stack uses FIFO delivery and receipt, causing a high-priority receiving task to potentially block until a lower priority task receives its message. Even with prioritized delivery such as EDF or priority-based delivery, priority inversion can occur if the protocol stack is implemented in the kernel. Suppose a deluge of messages for low-priority

tasks arrives at a node, the kernel processes these messages at kernel priorities (which is normally high) and possibly even uninterruptibly, thus causing other high priority tasks to starve. Several protocol processing software designs have been suggested by Lee et al. (1996) to minimize inversion. These include performing protocol processing using a shared communication protocol server, prioritized threads instead of a single high- or low-priority thread for message sending and receipt, and the application thread itself. Using a shared protocol server allows one to treat the server as a shared resource and apply the resource reservation model to it. Using prioritized threads allows preemption of low-priority protocol processing by high-priority ones. Application-level protocol processing amounts to implementing a user-library, which moves protocol processing to application space. BDM-RT is implemented as a user library with most of the host-side protocol processing embedded into the application's CPU time.

3.7 Real-time messaging on Myrinet

Myrinet has traditionally been used for high-performance distributed computing rather than for real-time communication, as discussed in Chapter 4.

FM-QoS (Connelly and Chien 1997) is the only known real-time messaging layer implemented on Myrinet. Of the essential components of real-time messaging layers discussed in this chapter, FM-QoS implements only a limited form of resource management. FM-QoS synchronizes link traffic to minimize switch contention (Connelly and Chien 1997) and consequently reduces LANai-to-LANai latency jitter. A brief description of FM-QoS is presented here. FM-QoS adopts a technique called Feedback Based Synchronization (FBS) to achieve self-synchronizing schedules that minimize resource conflicts. FM-QoS builds a global

view of time at the network interface by periodically sending self-synchronizing messages (Connelly and Chien 1997). These messages are intended to block one another at the Myrinet switch by virtue of having a common out-port. Based on the effect of this blocking on the network DMA latency, FM-QoS estimates the relative drifts of LANai clocks at the different nodes in the network to build a global view of time. This facilitates creating conflict-free schedules of link traffic.

The inception of extraneous self-synchronizing traffic introduces an overhead of up to 1% of the network bandwidth for an 8-node, 1-switch network. The overhead scales linearly with the relative drift rates between the LANai clocks. The scalability of this solution is questionable because of increased overhead in the presence of multiple Myrinet switches. FM-QoS is compared with BDM-RT in section 5.3.7.

3.8 Summary

To summarize, the design of real-time messaging subsystems needs to address the following main issues: predictability of message latency and protocol processing, management of shared resources, QoS sensitive processing of tasks, provision for best-effort traffic, traffic isolation, and time-bound and protocol processing free of undesired priority inversion. In a Myrinet network, switch contention and PCI bus contention need to be specifically addressed. FM-QoS is the only known implementation of a Myrinet real-time messaging layer, but falls short of meeting hard real-time requirements as it lacks explicit management of the PCI bus.

CHAPTER IV

HIGH-PERFORMANCE MESSAGING

This chapter highlights the goals of high-performance messaging systems and discusses the key design issues involved in achieving these goals. Although much of the discussion is applicable to high-speed networks in general, this chapter focuses chiefly on Myrinet-like high-speed hardware architectures. The network interface is assumed to have Direct Memory Access (DMA) capability to access host memory, and processing capability to offload protocol processing from the host CPU. Another underlying assumption is that raw network bandwidth is greater than that between the host and network interface, which is true of current high speed networking technologies such as Myrinet (Boden et al. 1995, 29), Giganet (Giganet Inc. 2000), and ServerNet (Horst and Garcia 1997). The following discussion applies in general to nearly all popular high-performance messaging layers over Myrinet such as GM (Myricom, Inc. 1999), FM (Pakin, Lauria, and Chien 1995), BDM (Henley et al. 1997), BDM/Pro (MPI Software Technology, Inc. 2000), and AM (von Eicken et al. 1992, 256). A comprehensive overview of the design of several high-performance communication systems on Myrinet is discussed by Bhoedjang, Ruhl, and Bal (1998, 53).

4.1 Design issues

Performance goals combined with the underlying hardware characteristics such as CPU speed, memory bandwidth, network bandwidth govern the messaging

software design. Performance goals can be classified chiefly into the following parameters (Bhoedjang, Ruhl, and Bal 1998, 53):

- Low message latency,
- High user-level bandwidth, and
- Minimal processor overhead.

The design is also influenced by other goals pertinent to high-performance distributed computing (Bhoedjang, Ruhl, and Bal 1998, 53) such as

- Degree of reliability,
- Scalability, and
- Support for multicast operations.

We now discuss the design choices available to messaging layers over a high-speed Myrinet-like network, given the abovementioned performance goals.

4.1.1 Host memory to network interface transfers

Host-to-host data transfers consist of three transfer stages: The sender's host memory to network buffer, the sender's network buffer to the receiver's network buffer, and the receiver's network buffer to host memory. Of these, two transfers occur between host and network memory. There are two ways for performing this transfer: Direct Memory Access (DMA), and Programmed I/O (PIO). The presence of a master-capable DMA engine on the Myrinet network interface creates the possibility of a host-initiated and MCP-initiated DMA. The effect of these design choices on performance are discussed below.

4.1.2 DMA vs. PIO

Fundamentally DMA is preferable to PIO for long transfers as the transfer does not require the host CPU, thus allowing overlapping of computation with communication. On a given hardware platform, if DMA is at least approximately as fast as PIO then DMA is the clear superior choice for achieving high bandwidth. However, on certain platforms PIO may be significantly faster than DMA. For example, figure 4.1 shows the DMA vs. PIO bandwidths on an UltraSparc with an SBus LANai interface. It is seen that Host-to-LANai PIO transfers are faster than DMA, for all transfer sizes. On this platform, BDM uses PIO for host-to-LANai transfers and DMA for LANai-to-host transfers (Henley et al. 1997). However, on a Pentium with PCI LANai interface DMA is superior to PIO as seen in figure 4.2. On this platform, BDM uses DMA for all transfers between host and LANai memory. According to Bhoedjang, Ruhl, and Bal (1998, 55) examples of systems that use DMA are VMMC (Dubnicki et al. 1997, 388), PM (Tezuka et al. 1998, 308), VIA on Giganet (Giganet, Inc. 2000), and GM.

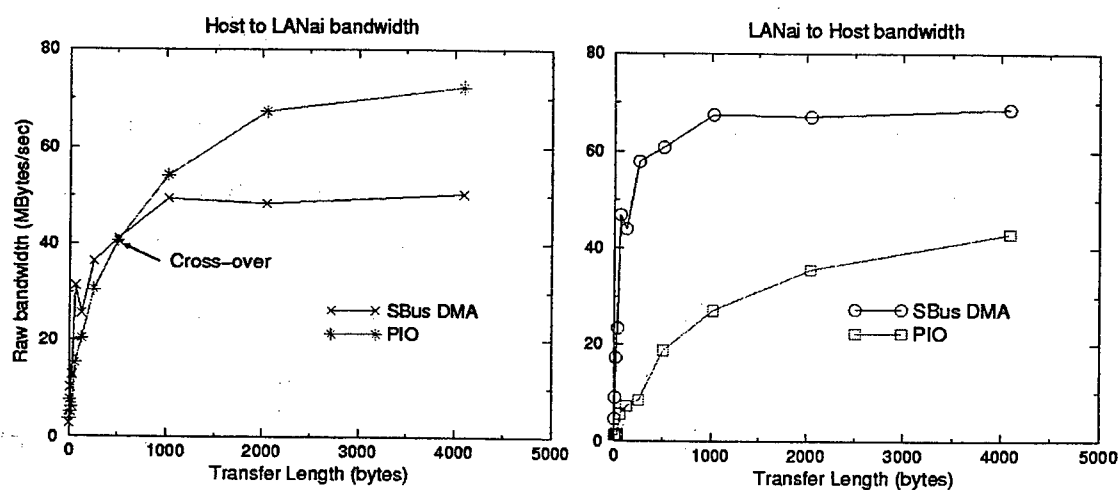


Figure 4.1: Raw SBus Bandwidth comparison on UltraSparc running Solaris

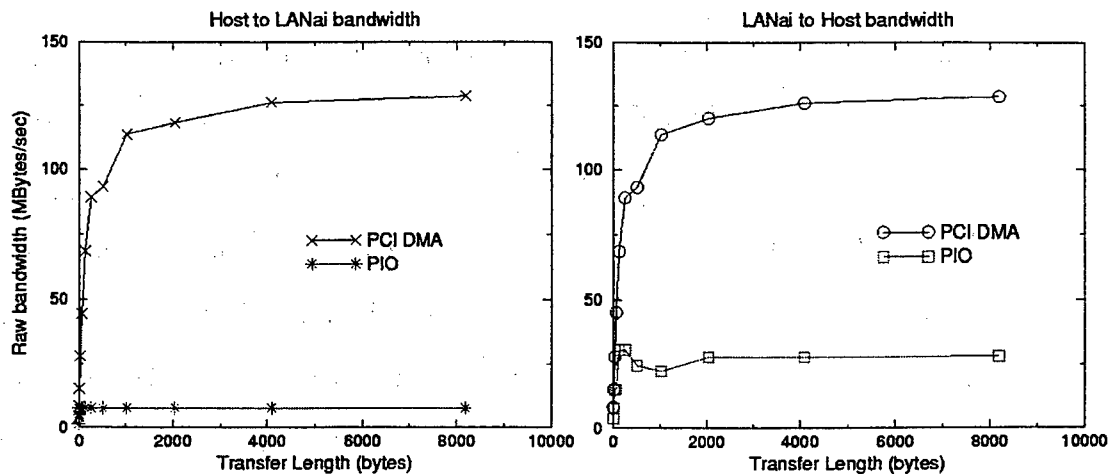


Figure 4.2: Raw PCI Bandwidth comparison on Intel x86 running Linux

DMA involves a fixed set-up cost that is absent in PIO. This normally causes DMA latencies to be higher than PIO latencies for short messages where the DMA setup cost is comparable to the actual transfer time. Consequently, high performance subsystems such as BIP (Prylli 1997) use a combination of DMA and PIO - with PIO for short transfers and DMA for long transfers.

4.1.3 DMA initiation and completion

On Myrinet, DMA is initiated by writing DMA parameters such as target addresses, DMA length and DMA direction into on-board LANai DMA registers. Because LANai memory is directly accessible by the host CPU, DMA initiation can be done either by the MCP or the host library. The same applies to detection of DMA completion, as it involves examining a LANai register value. Access to LANai registers by the host is across the inter-connecting bus (e.g., the PCI bus) and is significantly more expensive than access by the MCP. For this reason, it is more advantageous for the MCP to set up and complete DMA transfers between

the host and network interface. Another strong reason for avoiding the detection of DMA completion. Polling will then compete against the DMA transfer for bus cycles and affect performance.

4.1.4 Message receipt mechanism

Message receipt can be done either by polling or can be interrupt based. Interrupt based receives incur less processor overhead because the host processor does not waste CPU time to detect the arrival of a message. On the other hand, polling eliminates an extra context switch which is better for latency. On systems with high context switch overheads and high interrupt latencies, polling is the more attractive option. FM, BDM, and BIP (Prylli 1997) do polling based receives. Some systems also use a combination of polling and interrupts as an optimization for latency and processor overhead. GM, VMMC (Dubnicki et al. 1997, 388), and U-net (von Eicken et al. 1995, 303) are examples of such systems. GM allows user programs to choose the receipt mechanism by calling the appropriate receive function at run-time. GM also supports a hybrid receive mode where the user process polls for a finite duration before blocking on an interrupt.

4.1.5 Buffer queue management

Network buffer space is limited, and needs to be managed efficiently by messaging subsystems. The simplest buffer management scheme is to use FIFO queues to store used and free network buffers. BDM, GM and FM use FIFO queues for managing sent and received message buffers. Additional queues are implemented to store messages for potential retransmission if reliability is desired. FIFO queuing causes least queue-processing overhead, but at the same time

imposes the restriction of in-order delivery of messages unless a tag-based demultiplexing scheme is implemented at the receiver side. For example, in BDM, FM and GM out-of-order receives are not possible. However, GM has a provision for sending high and low priority messages, which allows out-of-order transfer to a certain extent. Typically, this is used by the distributed computing middleware such as MPI (Snir et al. 1995) as follows: the high priority channel is used to send small control messages while the low priority messages carry actual data.

4.1.6 Zero-copy mechanism

Many messaging layers improve latency and bandwidth by avoiding making extra copies of user data within host memory itself. If DMA is used, this requires the user's data to reside in DMA-able memory pages (to be pinned in physical memory and not swapped out by the Virtual Memory subsystem). Two ways of achieving zero-copy is by pinning user pages to memory (if the Operating System allows this) or by mapping pinned kernel pages to user space. GM supports both methods of allocation by providing functions to both allocate a DMA-able region, as well as to pin a portion of user's memory. On Linux, BDM allocates kernel pages and maps them to user space at initialization time.

4.1.7 Reliability

Nearly all Myrinet messaging layers provide reliable communication. The Myrinet Link layer itself provides a relatively high degree of reliability using a CRC. Further, bit errors are as low as 1 in 10^{15} bits transmitted. However, reliability is desired to recover from receive buffer overflows. GM and FM provide a reliable, ordered communication protocol. BDM provides a multi-protocol suite ranging

from unreliable communication to reliable and ordered communication. Reliability reduces performance because of the additional overhead of ACK/NACK packets. However, low-level messaging provide reliability as it is a basic requirement in the high-performance computing environment (Snir et al. 1995) (Sunderam et al. 1994, 531) and is too costly too implement in higher layers.

4.2 Timeliness and resource management

Despite design differences between high-performance messaging layers such as BDM, FM, GM, AM, VMMC, BIP, and U-Net, a common aspect of their design is the absence of explicit mechanisms to ensure timeliness of message transfer in the presence of contending traffic. These messaging layers are incapable of providing specific guarantees on bandwidth and latency. It is seen that minimizing resource contention, performing QoS sensitive processing, and providing bounded-time message delivery do not appear among the goals of high-performance messaging layers. In this respect, BDM is similar to all other high-performance layers cited in this chapter.

The insufficiency of high-performance layers for real-time communication is best illustrated by discussing a relevant MPI/RT development effort by the author of this thesis. Time-based MPI/RT Channel (Kanevsky, Skjellum, and Watts 1997) was recently implemented over BDM/Pro (MPI Software Technology, Inc. 2000) on a multi-computer platform. BDM/Pro is a high-performance messaging layer that performs low-latency and high-bandwidth communication on certain Myrinet based multi-computer platforms. For performing time-based communication, a global clock was implemented by the MPI/RT messaging layer. The achieved clock accuracy was of the order of 1 millisecond because of the absence of timeliness

of clock message delivery by BDM/Pro. Compared to the microsecond range latencies of messages on Myrinet (Boden et al. 1995, 29) this is unacceptable granularity for time-based communication. In contrast, the BDM-RT based global clock (see section 2.2.1) achieves granularity that is realistically usable for real-time communication.

4.3 Summary

In summary, high-performance messaging layers share the common goals of low latency, high bandwidth, low processor overhead. The design of these layers is influenced strongly by their goals. On a Myrinet-like architecture, various design options exist with respect to activities such as the transfer mechanism between host and network buffers, message receipt mechanism, zero-copy mechanism, and level of reliability. The hardware platform and Operating System parameters generally govern the choice of these mechanisms. With respect to timeliness of message delivery and QoS sensitive protocol processing, BDM can be considered as representative of the other messaging layers because of the absence of real-time components in any of the high-performance messaging layers.

CHAPTER V

DESIGN OF BDM AND BDM-RT

This chapter describes the design of BDM and BDM-RT. The key aspects of their design has been discussed and justified, keeping in view their differing goals of performance and predictability. This chapter is a precursor to the next chapter that discusses the impact of BDM's design on its predictability and that of BDM-RT's design on its performance.

5.1 Messaging Protocol

Both BDM and BDM-RT use the BDM unreliable protocol (Henley et al. 1997) for message transfer, in keeping with Real-Time Channel semantics (Indiresan, Mehra, and Shin 1995). A certain degree of reliability is achieved by the reliable Myrinet link layer which offers bit-errors as low as 1 error in 10^{15} bits transmitted. The protocol is simple and involves no handshake, acknowledgment, or flow control. When data is ready to be sent, the MCP at the sender's node initiates a network send DMA (sDMA). At the receiver's node, the MCP detects the incoming packet and issues a network receive DMA (rDMA).

5.2 BDM design

BDM for PromisQoS evolved from porting the original BDM software (Henley et al. 1997) for Solaris. However, certain aspects of its original design have been changed to exploit the differences between the two Operating Systems and between

Intel and UltraSparc hardware architectures. The porting and design changes were implemented as a part of this thesis work.

5.2.1 BDM queues

BDM maintains five queues to buffer incoming and outgoing messages. These queues are shown in the table 5.1, with a brief description of each queue. The queues SBQ, SLQ, RBQ, RLQ, and WAQ are similar to their counterparts in the original BDM (Henley et al. 1997), but are not identical in terms of the status of user data associated with buffers in these queues. The elements in these queues contain pointers to free or filled LANai buffers. In addition, they contain information about each LANai buffer's "shadow" buffer on the host. Shadow buffers are DMAable buffers on the host that have a one-to-one mapping with every LANai buffer. Shadow buffers are used by BDM to transfer data directly to or from LANai memory via PCI DMA.

5.2.2 Message flow

To send a message, the sender first allocates a free buffer using `BDM_Frame_malloc()`. This function provides the user with a handle to a free LANai send buffer and its DMAable "shadow" buffer on the host from SBQ. After filling the shadow buffer with user data, the function `BDM_Frame_send()` is called. This causes the send buffer and its shadow buffer to be placed in SLQ. The MCP in its mainloop pulls this buffer off the SLQ and initiates a PCI DMA to transfer user data from the shadow buffer in the host to its corresponding LANai buffer. Upon completion of the DMA transfer, the MCP initiates another DMA (sDMA) to send the data into the network. After the data has been completely transferred

Table 5.1: BDM Queues

Queue	Queue Name	Description	Producer	Consumer
SBQ	Send Buffer Queue	Queue of free send-buffers	MCP	Host
SLQ	Send LANai Queue	Queue of send-buffers with valid user data in the "shadow" buffers on the host	Host	MCP
RBQ	Receive Buffer Queue	Queue of free receive-buffers	Host	MCP
RLQ	Receive LANai Queue	Queue of receive-buffers with valid data in the LANai buffer	MCP	Host
WAQ	Wait for Ack Queue	Queue of filled send-buffers that have been already sent by a reliable protocol and awaiting an acknowledgment from the receiver	MCP	MCP

out the network, the LANai buffer - shadow buffer pair is put back into SBQ. Figure 5.1 shows the various states of a send buffer.

On the receiver's side, the MCP allocates a receive buffer from RBQ and checks for arriving network traffic in its mainloop. Upon detection of message arrival, the MCP performs a network receive DMA (rDMA) to transfer data into LANai buffer space. The message is then transferred to the LANai buffer's shadow location on the host by initiating a PCI DMA transfer. Upon completion of this DMA, the MCP puts the buffer pair into RLQ. The host program calls `BDM_Frame_rcv()` to receive this message, and subsequently frees the buffer into RBQ by calling `BDM_Frame_free()`. Figure 5.2 shows the various states of a receive buffer.

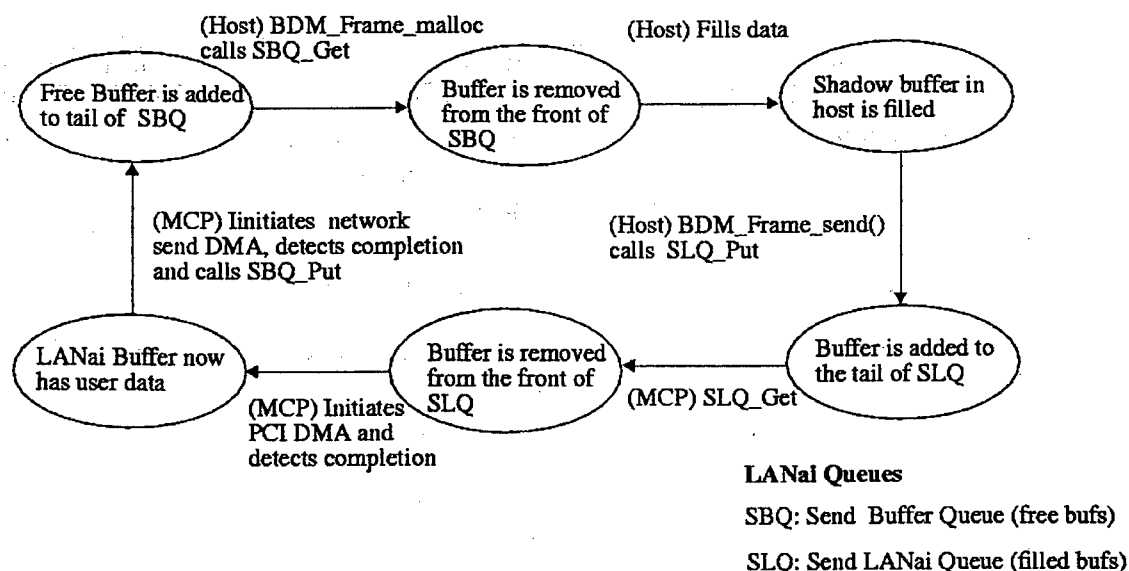


Figure 5.1: BDM Send Queues

5.2.3 Message receipt

5.2.3.1 Polling vs. interrupt-based

BDM uses polling to detect the receipt of messages. The trade-offs involved between polling and interrupt based receipt has been discussed in section 4.1.4. In brief, BDM uses polling to tradeoff processor overhead for better latency.

5.2.3.2 The receive function

BDM messages are received via the function `BDM_Frame_rcv()`. A call to `BDM_Frame_rcv()` looks for a received message that has already been transferred to the host. The BDM MCP transfers all received messages to the host as soon as they are received. A call to `BDM_Frame_rcv()` succeeds if there is at least one message in a shadow receive buffer on the host.

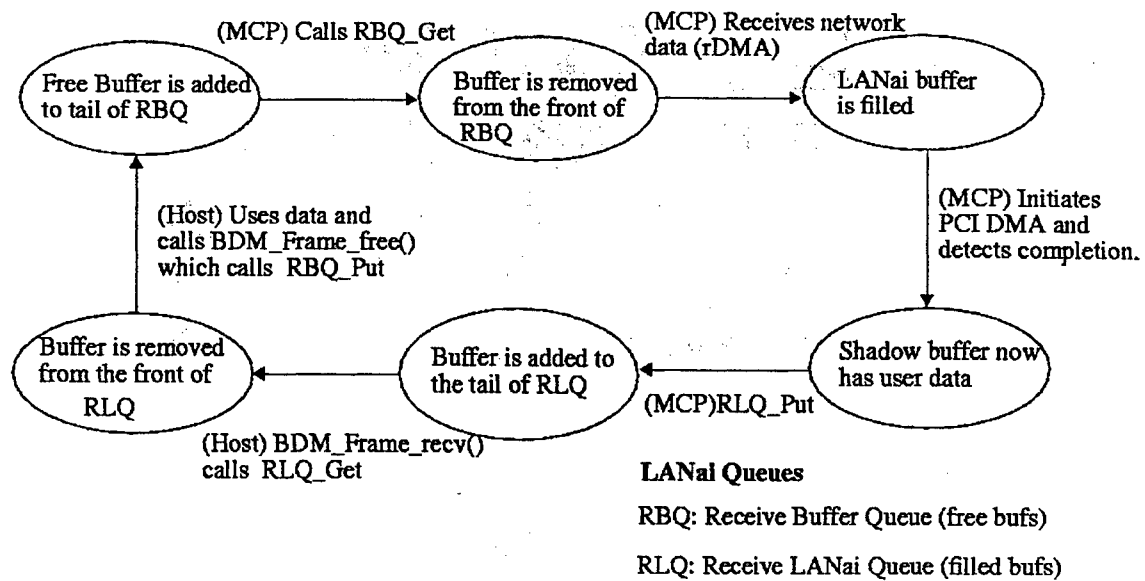


Figure 5.2: BDM Receive Queues

Rationale: In this design of message receipt, the MCP transfers incoming data to shadow buffers on the host in a best-effort fashion without waiting for the host to call the receive function. An alternative design is to cause `BDM_Frame_rcv()` to look for messages received and buffered in the LANai itself, and then request the MCP to initiate a DMA transfer to the host. The receive function should then detect DMA completion and return. The chosen design is better as compared to the cited alternative for the following reasons: The message latency is better because we do not need any LANai memory accesses across the PCI bus either to detect a received message or to initiate PCI DMA transfer or detect its completion. Latency is also reduced by minimizing the time spent by received messages in the LANai buffers before being transferred to the host. Lastly, we get better bandwidth by avoiding blocking in the receive function for the duration of the PCI DMA transfer.

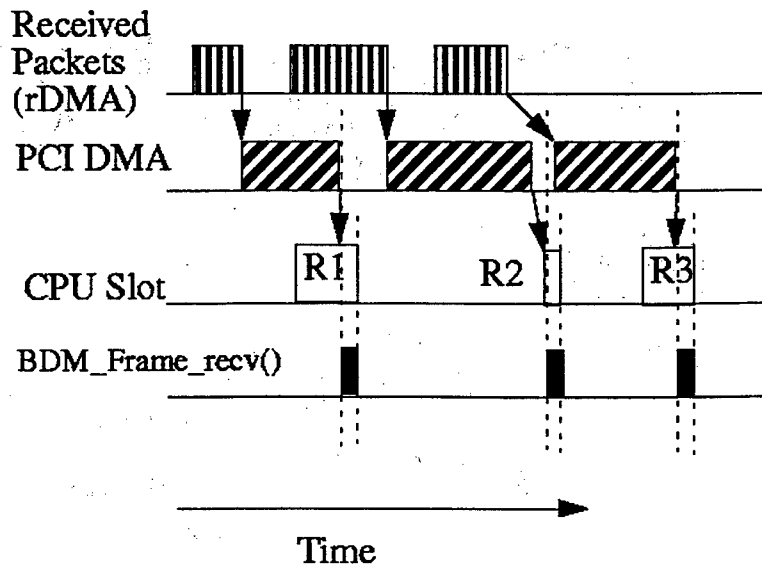


Figure 5.3: Receive Semantics for BDM

5.2.4 Host-LANai data transfer

BDM uses PCI DMA rather than PIO for all transfers between host and LANai memory. Initiation and detection of completion of DMA transfers is done by the MCP.

The choice of performing DMA over PIO is best for bandwidth because of superior PCI DMA rates as compared to CPU assisted copy on our architecture 2.1. The bandwidth for both DMA and PIO are shown in figure 4.2. Because of superior bandwidth for both reads and writes, PCI DMA is used at both the sender's and receiver's end. However, it must be noted that PIO is better than DMA for short message latency, because of the absence of fixed DMA set-up overheads. Thus, it is best to use a poly-algorithm that chooses between PIO and DMA based on the message size.

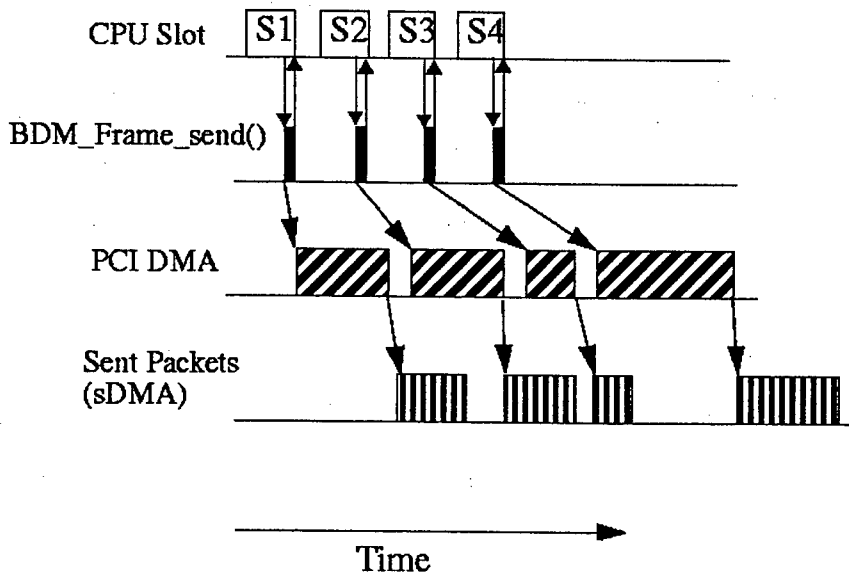


Figure 5.4: Send Semantics for BDM

5.2.5 Message receipt order

All queues in BDM are FIFOs. `BDM_Frame_rcv()` returns messages in the order they arrived at the LANai. No tag-based de-multiplexing is implemented, as it not directly related to its primary goal of reliable and efficient communication. Such a de-multiplexing mechanism can always be implemented at a higher layer if desired by applications, though at some cost of performance.

5.3 BDM-RT design

This section outlines the key design aspects of BDM-RT, given its goals of predictable and QoS-sensitive communication. Performance is also important, but is a secondary goal.

5.3.1 BDM-RT queues

BDM-RT maintains three buffer queues for managing incoming and outgoing messages – SBQ, SLQ, and RBQ. These are identical to the ones in BDM described in 5.2.1. RLQ is replaced by DLL - a linked-list of filled LANai buffers. WAQ is absent because of the unreliable protocol used in Channel communication. BDM-RT also maintains another linked-list DBL, which is a list of free structures containing information useful for PCI DMA in addition to a pointer to a free buffer from RBQ. These queues and lists are shown in table 5.2, with a brief description of each entity.

Table 5.2: BDM-RT Queues and Lists

Queue	Queue Name	Description	Producer	Consumer
SBQ	Send Buffer Queue	Queue of free send-buffers	MCP	Host
SLQ	Send LANai Queue	Queue of send-buffers with valid user data in the "shadow" buffer on the host	Host	MCP
RBQ	Receive Buffer Queue	Queue of free receive buffers	Host	MCP
DBL	DMA Buffer List	Linked list of free receive buffers with DMA information	MCP	MCP
DLL	DMA LANai List	Linked list of LANai receive buffers with valid user data ready to be transferred to the host upon request	MCP	MCP

5.3.2 Message flow

On the send side, the message flow is similar to that described in BDM design in 5.2.2. The only difference on the send side is that the send function `BDMRT_Frame_send()` returns only after waiting for the sent message to be transferred into LANai memory. Figure 5.1 shows the various states of a send

buffer. On the receiver's side, the MCP allocates a receive buffer from RBQ and a DMA-info element from DBL and checks for arriving network traffic in its mainloop. After receiving data, the MCP fills the received tag information and buffer address into the allocated DMA-info element and places it in DLL. When the host calls `BDMRT_Frame_rcv()` with a tag, the MCP searches DLL for the earliest message received with the specified tag. If found, the message is transferred to the host buffer via PCI DMA. Upon completion of this DMA, the MCP puts the DMA info element into DBL and signals the host by setting a shared flag. The call to `BDMRT_Frame_rcv()` returns successfully and returns a pointer to the received message. If a message with the specified tag is not found, the MCP signals failure information to the host, and `BDMRT_Frame_rcv()` returns failure. After a successful receive, the host frees the buffer into RBQ by calling `BDMRT_Frame_free()`. Figure 5.5 shows the various states of a receive buffer.

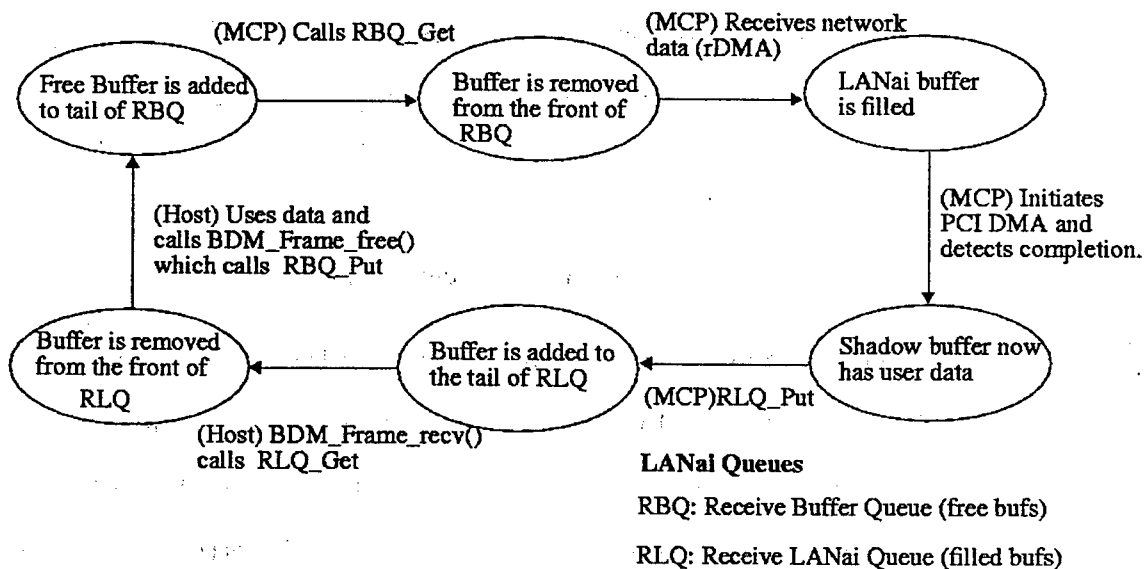


Figure 5.5: BDM-RT Receive Queues and Lists

Rationale: The message flow is based on synchronizing local resource accesses with the local CPU schedules. Network protocol processing is tightly coupled with the CPU schedules to minimize contention for locally shared resources. The message flow is also designed to minimize contention of globally shared resources - namely the switch and network bandwidth - by adopting globally synchronized message transfers. Notice that by synchronizing the send-side host-to-LANai DMA transfer with the sending task's CPU schedule, and the receive-side transfer to the receiving task's CPU schedule we use a mixture of "push" and "pull" models of communication. At the sender's node, the data is "pushed" to the local LANai buffers in accordance with its schedule, while the data is "pulled" out of the receiver's local LANai buffers only after the process is scheduled locally on the CPU. The key aspect, of course, is that predictable message transfer between the two LANai interfaces has to be ensured. This is achieved by globally scheduling all contending senders across the network using BDM-RT's fine grain global clock. Combined with bounded-time protocol processing at the MCP, this will ensure that no two messages with the same destination port will arrive at a switch at the same time.

5.3.3 Message receipt

BDM-RT uses polling rather than interrupts in keeping with the philosophy of the time-based real-time Operating System PromisQoS. In a time-based system, all events such as message receipt, are expected to occur at their scheduled time-intervals which precludes asynchronous events such as interrupts. Message receipt at the host occurs as follows: A call to the receive function `BDMRT.Frame.recv()` sets a flag on the LANai. Upon detecting this set flag, the MCP searches the

list of received messages on the LANai for a received message that matches the requested tag. Then the MCP sets another shared flag to indicate the result of the search, and initiates a PCI DMA transfer to the host if found. If found, the host waits for the estimated duration of PCI DMA before polling for DMA completion at regular intervals of 5 μ sec. If no matching message was found, the function `BDMRT_Frame_rcv()` returns failure.

Rationale: The three main aspects of this receive semantics are as follows: PCI DMA occurs only after a call to `BDMRT_Frame_rcv()`, the MCP initiates and finalizes DMA transfers, and the `BDMRT_Frame_rcv()` function blocks the host CPU until the transfer is complete. The rationale for performing PCI DMA only upon a call to `BDMRT_Frame_rcv()` is to couple the PCI bus resource to the host CPU, which itself is managed by a real-time scheduler. This coupling automatically leads to a QoS-sensitive management of the PCI bus as it becomes a controlled resource (Lee et al. 1996) with the host CPU as the controlling resource (Lee et al. 1996).

For better overall latency, the MCP initiates DMA and detects its completion, instead of the host library. However, latency jitter is introduced because of the time that elapses between the host setting the ready flag and the MCP detecting the set flag in its mainloop. This jitter component is absent if the host directly initiates DMA by writing to LANai DMA registers. The rationale for choosing the MCP to initiate DMA is that the maximum jitter introduced by this was empirically measured to be less than the latency overhead introduced by the host directly initiating DMA. A DMA transfer requires the writing of four LANai registers: the source address, the destination address, the DMA direction, and the number of bytes to transfer. The latency of a single LANai DMA register read/write

operation across the PCI bus (i.e., from the host) is as high as 5 μ sec, while it is less than 0.1 μ sec when done from the MCP. Thus if the host sets up a DMA, we incur a 20 μ sec latency increase. On the other hand, the maximum measured interval between two successive polling operations in the MCP to detect if the host has set the ready flag is approximately 6-8 μ sec, which is a clear performance gain.

BDM-RT performs what we have termed as “blocking DMA” transfers between the LANai and Host. The host CPU is blocked by the receiving BDM-RT task during the entire duration of a PCI DMA transfer. The main reason for this is to disallow potential PCI bus users from being scheduled on the CPU during an on-going Myrinet PCI DMA. This greatly reduces PCI bus contention from non-real-time Linux applications serviced by the Ethernet and SCSI drivers, leading to an improvement in DMA latency predictability.

“Blocking DMA” represents a classic tradeoff between performance and predictability. CPU utilization and overlap of computation and communication are traded off for better predictability of DMA latency. For short transfers, the blocking duration is acceptably low (≈ 5 μ sec for transfers up to 512 bytes). For long messages, this involves greater wastage of CPU time (≈ 62 μ sec for 8000 bytes). However, compared to Programmed I/O (≈ 280 μ sec for 8000 bytes), “blocking DMA” still consumes significantly lesser CPU time.

5.3.4 Host-LANai data transfer

BDM-RT uses PCI DMA rather than PIO because performance is also a goal. BDM-RT uses “blocking DMA” semantics for both LANai-to-host and host-to-LANai PCI DMA. A call to `BDMRT_Frame_send()` sets a ready flag on LANai memory. When the MCP detects this flag, it initiates a DMA

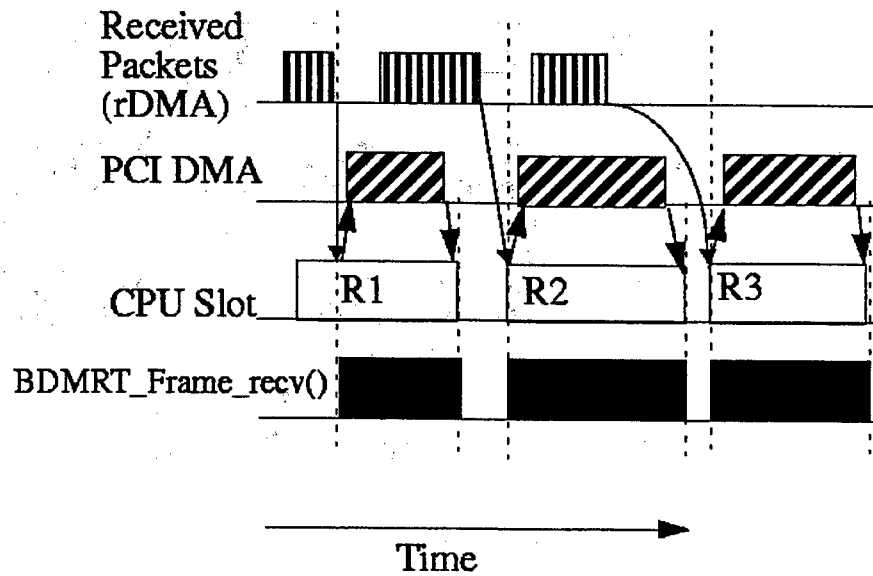


Figure 5.6: Receive Semantics for BDM-RT

transfer. `BDMRT_Frame_send()` returns after holding the CPU idle for the estimated average duration of PCI DMA for the given message length (calculated empirically). Note that although this is similar to the blocking nature of the receive function, `BDMRT_Frame_send()` does not wait until the actual completion of the DMA unlike `BDMRT_Frame_rcv()`. Figure 5.7 illustrates the time-line of activities associated at the send side.

Rationale: “Blocking DMA” improves the predictability of DMA latency and consequently the predictability of the overall message latency as discussed for receive side LANai-to-host PCI DMA. The reasons for avoiding blocking until the actual completion of host-to-LANai DMA are (a) to avoid extra processor overhead associated with the host detecting the DMA completion, and (b) to avoid the ill-effects of polling (for detecting DMA completion) on the jitter of the PCI DMA latency. `BDMRT_Frame_send()` instead blocks until the average duration of PCI

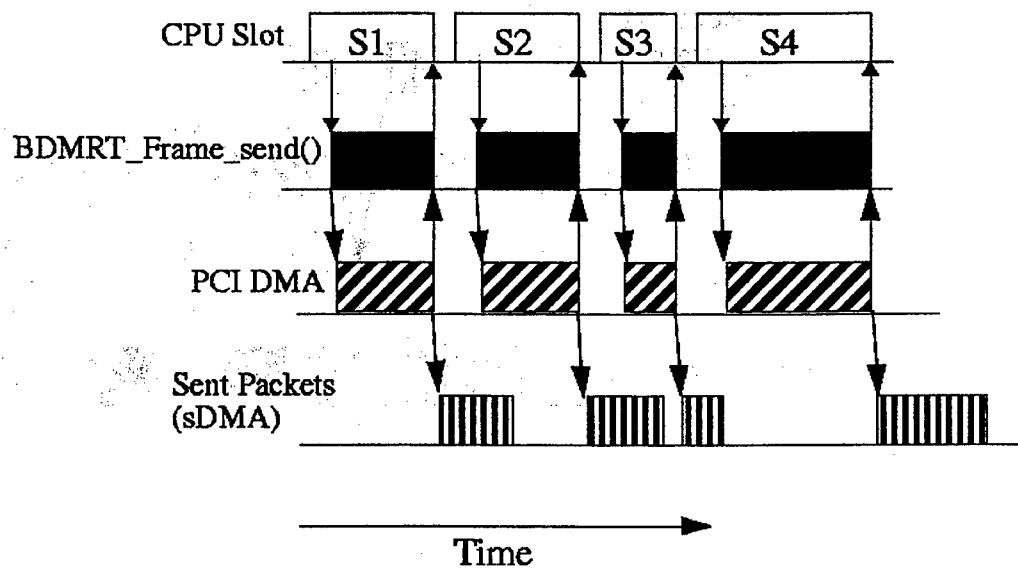


Figure 5.7: Send Semantics for BDM-RT

DMA latency elapses. In the worst case, BDMRT_Frame_send() still polices the major part of the PCI DMA transfer even if the actual transfer duration exceeds the estimated average.

5.3.5 Resource management

The chief resources required to be managed by BDM-RT to provide QoS-sensitive communication to various real-time tasks are listed below.

- Network bandwidth,
- PCI bandwidth,
- LANai buffer space, and
- LANai CPU time.

The following sections describe how each of these resources is managed by BDM-RT.

5.3.5.1 Network bandwidth

Network bandwidth deprivation is mainly caused when a message blocks at the switch because of contention from another message with the same out-port. Network bandwidth is managed by preparing message schedules based on a fine-grain global clock. BDM-RT provides a mechanism for high-quality transmission of periodic clock synchronization messages, to facilitate the implementation of a high accuracy global clock. This was described in section 2.2.1.

5.3.5.2 PCI bandwidth

The LANai4.x PCI DMA engine can perform one DMA transfer at a time, in half-duplex fashion¹. Thus bus contention can exist between sending and/or receiving BDM RT tasks. PCI bandwidth is managed among BDM-RT tasks by performing PCI DMA transfers during a task's CPU time. This totally eliminates PCI bus contention between BDM-RT tasks, and greatly reduces the effect of bus contention from the PCI ethernet device transferring non-BDM-RT best-effort Linux traffic.

5.3.5.3 LANai buffer space

Ideally, it is desirable to allocate a separate buffer pool for each MPI/RT channel task. Because of limited memory on the LANai interface, this solution does not scale well, although it provides good resource isolation. LANai buffer

¹LANai5.x and higher change this to allow two DMAs, only one progressing per unit time.

space is organized in send and receive queues as described in section 5.3.1. The limit on LANai buffer space imposes an extra constraint on message scheduling in order to ensure that the receiver's buffers do not overflow at any point in time. This is not a trivial problem, particularly when one scales to larger networks.

5.3.5.4 LANai CPU time

LANai CPU time is spent mainly between protocol processing and initiation and completion of DMA transfers to and from the host and network. The LANai processor has its own 32-bit Real Time Clock that is not synchronized to the host clock. Because of the absence of a view of global time, the MCP does not attempt to schedule protocol processing or DMA activity. Instead, it provides bounded response time to all events by avoiding any blocking activity in its mainloop. For example, the MCP never blocks on any type of DMA to ensure that other messages waiting to be processed are serviced with bounded latency.

5.3.6 Priority inversion

BDM-RT avoids priority inversion by (a) embedding most of the protocol processing overhead in the tasks' CPU time and by (b) allowing out-of-order receipt of messages.

5.3.7 Comparison with FM-QoS

FM-QoS (Connelly and Chien 1997) was mentioned as the only known implementation of a real-time messaging layer on Myrinet and was described in section 3.7. To highlight BDM-RT as one the significant contributions of this

thesis work, BDM-RT is compared to FM-QoS with respect to predictability and performance overhead.

While FM-QoS guarantees bounded latencies between LANai interfaces, BDM-RT goes a step further to guarantee bounded end-to-end latency. This is a result of explicit PCI bus management and bounded MCP response-time in BDM-RT. BDM-RT and FM-QoS differ in their approaches to provide global synchronization. BDM-RT tightly couples link schedules with CPU schedules that are based on a global clock (see section 2.2.1) at the host. On the other hand, FM-QoS de-couples the network processor from the host processor and schedules link traffic based on a global view of time at the network interface. Consequently, FM-QoS facilitates conflict-free link traffic schedules, while BDM-RT facilitates conflict-free end-to-end message schedules.

FM-QoS views network bandwidth as a series of slots (Connelly and Chien 1997), each of which has to be fully allotted to a single process (or channel). The granularity of clock synchronization in FM-QoS is dependent on the chosen duration of slots. For a realistic slot duration of 12 μ sec (Connelly and Chien 1997), self-synchronizing traffic can have periodicities between 12 and 36 milliseconds (Connelly and Chien 1997) and can yield accuracies of about $\pm 6 \mu$ sec (half the slot duration). In comparison, BDM-RT implements a global clock accuracy of $\pm 4 \mu$ sec with re-synchronizing periods as large as 1 second. In addition, re-synchronization traffic in BDM-RT consists of 12-byte sized clock messages compared to significantly longer self-synchronizing messages in FM-QoS (e.g., 2 KB sized messages for 12 μ sec slots). Consequently, the overhead for BDM-RT clock synchronization is significantly lesser than that reported for FM-QoS by Connelly and Chien (1997). For example, on a 8-node, 1-switch network with clock

drifts of 200ppm, BDM-RT consumes less than 140 μ sec of link time every second amounting to an overhead of $\approx 0.014\%$. FM-QoS incurs an overhead of 0.32% (Connelly and Chien 1997) for the same configuration and comparable accuracy.

The overhead of the FM-QoS synchronization technique for multiple-switch networks is not clear from the available literature. Self-synchronizing schedules become complex in the presence of multiple switches (Connelly and Chien 1997). Such a problem does not with BDM-RT arise because of the relative simplicity of the master-slave synchronization scheme adopted by the global clock algorithm. Although the synchronization overhead in BDM-RT increases linearly with the number of nodes in the network, the absolute overhead on link bandwidth works out to be less than 0.2% for networks with as many as 100 nodes.

5.4 Summary

The key design components of BDM and BDM-RT were presented in this chapter. A rationale was provided for all adopted design choices in BDM and BDM-RT.

BDM was ported to PromisQoS as a part of this thesis work to study its performance and predictability. BDM performs strictly FIFO queue processing to minimize overhead. The message flow in BDM involves PCI DMA transfers initiated by the MCP in a work-conserving fashion. The send function sets a flag on the LANai and returns, while the actual PCI DMA transfer takes place when the MCP detects the set flag. The receive function `BDM_Frame_recv()` returns the first message that has been fully transferred into host memory. These design choices are governed by the performance goals of low latency, low processor overhead, and high bandwidth.

BDM-RT was designed and implemented as a part of this thesis work to provide hard real-time communication for MPI/RT Time-based Channels (Kanevsky, Skjellum, and Watts 1997). It is the first known hard real-time messaging layer on Myrinet. BDM-RT delivers messages in FIFO order, but allows for a tag-based demultiplexing scheme at the receiver's end. PCI DMA activity in BDM-RT is synchronized with the send and receive functions that block during DMA transfers. BDM-RT has explicit support for high-quality transmission of clock messages to allow the implementation of a fine grain global clock. The abovementioned design components improve the predictability of BDM-RT message latency and minimize contention of shared resources.

A comparison of BDM-RT with the only other real-time Myrinet messaging layer FM-QoS shows that BDM-RT goes beyond FM-QoS to provide predictable end-to-end latency. Additionally, the global clock overhead of BDM-RT is about an order of magnitude lesser than the synchronization overhead incurred by FM-QoS for comparable accuracies.

Having presented the detailed design of BDM and BDM-RT, the next chapter analyzes their design differences with respect to predictability and performance.

CHAPTER VI

COMPARISON OF BDM AND BDM-RT

This chapter demonstrates the fundamental dichotomy between the low-level design of a real-time messaging layer and that of a high-performance messaging layer. BDM, which was designed for performance, has been analyzed with respect to its unsuitability for real-time message transfer. The various components of BDM that primarily enhance performance but negatively impact time-lines are discussed. Likewise, aspects of design in BDM-RT that enhance its predictability are evaluated for their performance. In this chapter we compare and contrast design choices made for BDM and BDM-RT and their effect on performance and timeliness of message delivery. It should be noted that BDM-RT evolved from BDM by modifying its design to make BDM-RT predictable and QoS-sensitive.

6.1 Performance-predictability trade-offs

This section gives some background on the requirement of trade-offs between predictability and performance for messaging layers built on COTS platforms. Games et al. (1995) have addressed the question of whether general purpose commercial massively parallel processors (MPPs) can be used for computationally intensive real-time applications. The study focused around real-time scheduling of communication between processing nodes, and providing the desired predictability without undue sacrifice of performance (Games et al. 1995). Myrinet clusters of general purpose desktops bear strong resemblance to MPPs in terms of the

underlying network characteristics. It is clear from the analysis of MPPs by Games et al. (1995) that the high-capacity of Myrinet-like networks does not necessarily translate into predictable communications performance.

Although performance trade-offs are not always necessary for real-time communication, they are often required because of hardware architectural constraints. As is the case with Myrinet and with COTS platforms, the hardware offers minimal or no provisions for timely accesses to resources. Consequently, it becomes the responsibility of real-time software to make appropriate design choices potentially at the cost of performance. As an example, consider a hypothetical activity of performing periodic DMA transfers between host and LANai memory at a Myrinet interface. For best predictability, the transfer should occur during a scheduled time-interval based on the host's APIC timer value. Using the on-board LANai DMA engine for DMA transfers, there is a choice between using the host or the MCP to initiate the transfer. For best timeliness, the host should initiate the DMA transfer because the host can access APIC timer values much more accurately than the MCP. But with regard to performance, this degrades message transfer latency because access to DMA registers from the host is much slower than access from the MCP. Here, the conflict between performance and timeliness arises because of the unavailability of a fine-grain clock on the LANai board.

As a second example, access priorities (Myricom Inc. 1996) govern accesses to LANai memory by the LANai processor, by the host processor, and by the on-board DMA engines. In the absence of other PCI bus contenders, the access-time for the host processor to LANai memory is most predictable as it is of highest priority, but is also the slowest as it involves an access across the PCI bus. This makes accesses from the host more predictable, but poor in terms of performance.

Modern systems provide a good level of flexibility by supporting multiple configurations of system parameters at the BIOS level. Appendix A contains a discussion on exploiting the programmability of PCI configuration registers (Solari and Willse 1998) and other hardware solutions to improve the predictability of the system. None of the hardware/firmware solutions proposed in appendix A were adopted as they are highly dependent on the adherence of device manufacturers to the PCI standard and sometimes require specialized hardware.

6.2 BDM and BDM-RT: Design differences

In this section, the chief components of BDM and BDM-RT are compared and contrasted. The analysis of BDM and BDM-RT design is organized as follows: For each design component, the choice of design adopted by BDM is first described and then justified by providing the design rationale. The design is then evaluated with respect to its (un)suitability to predictable message passing. On similar lines, each design component of BDM-RT is described and justified. The design is then evaluated with respect to its effect on the performance of the system. It will be shown that in most cases, BDM is unsuited for real-time communication and that BDM-RT makes performance tradeoffs to achieve the desired predictability.

3.2.1 MCP main-loop

As stated in chapter 2, the Myrinet Control Program (MCP) runs on the LANai processor and buffers data going out of the host and coming in from the network. An MCP typically runs in an infinite loop (the main-loop), handing off to and receiving buffers from the host library and the network.

6.2.1.1 BDM

A simplified pseudo-code for the BDM-MCP main-loop is shown below:

```
1 while (1)
2     do
3         /* Sending messages */
4         if (An initiated network send DMA has completed) or
5             (No network send DMA was initiated)
6             then
7                 /* Sending: Buffer management */
8                 if (An initiated network send DMA has completed)
9                     then
10                        Put buffer back into free-send-buffer-queue
11                    fi
12
13                 /* Sending: To Network*/
14                 if (An initiated Host→LANai PCI DMA has completed)
15                     then
16                        Start DMAing the buffer out the network
17                    fi
18
19                 /* Sending: Host→LANai */
20                 if (No Host→LANai PCI DMA is in progress) and
21                     (Host has queued any buffers in send-buffer-queue)
22                     then
23                    Wait for any ongoing LANai→Host PCI DMA
24                    and Hand buffer to host. /* Blocking */
25                    Initiate Host→LANai PCI DMA for the first
26                    buffer in the send queue.
27                fi
28            fi
29
30        /* Receiving: From network */
31        if (Message is waiting to be received from the network)
32            then
33                Initiate receive-DMA
34                Loop until DMA finishes /* Blocking */
35
36        /* Receiving: LANai→Host */
```

```

37         if (LANai→Host or Host→LANai PCI DMA is in progress)
38         then
39             wait for PCI DMA completion /* Blocking */
40         fi
41         Initiate LANai→Host PCI DMA.
42     fi
43
44     /* Receiving: LANai→Host */
45     if (An initiated LANai→Host PCI DMA has completed)
46     then
47         Put buffer into received-msg-queue on host
48     fi
49 done /* while */

```

Rationale: As is evident from the pseudo-code, the MCP essentially manages sending and receiving data using the three on-board DMA engines and buffer queues on LANai buffer space. BDM-MCP is designed for minimal message latency. Highest priority is given to servicing received messages so as to minimize receive-buffer overflows and consequent re-transmission leading to performance-loss. At three points in the code (commented as "blocking"), the MCP waits on the completion of DMAs. A blocking network receive DMA is used to reduce message latency. The other two waits for completion of previously initiated PCI DMA transfers are necessary to avoid starvation of LANai→Host traffic in the presence of potentially too many back-to-back Host→LANai transfers and vice versa. (Access to the PCI bus is basically half-duplex because a single DMA engine that manages both directions). BDM uses buffer queues on the LANai with "shadow buffers" allocated in DMAable memory on the host. For every buffer on LANai memory, there is a corresponding "shadow buffer" on the host that goes through the same states as the original buffer. The notion of shadow buffers simplifies queue management between MCP and the host library. When a message

is received into a LANai buffer it is DMA-ed to its shadow location, after which it becomes available to the host (upon a call to `BDM_Frame_recv()`). The MCP initiates and winds up PCI DMA transfers, instead of the host library, because access latency for DMA engine registers (on-board) is much lower from the MCP as compared to that from host CPU.

Effect on predictability: The three segments of code where the MCP waits on the completion are clearly detrimental to the predictability of protocol processing latency and message latency in general. Considerable jitter in message latency can result because the MCP may be busy waiting on the DMA completion of another packet. For example, if the host puts a message into the send-buffer-queue and the MCP has just begun to DMA a received message, significant time will have passed before the MCP detects the presence of this buffer in the send-buffer-queue. Let us say, the host sends a 100 Byte message, and that an 8KByte packet arrives exactly when the host places the buffer in the send-buffer-queue. The MCP can spend as much as 50 μ sec on the blocking receive DMA (line 34 of pseudo-code) assuming a DMA bandwidth of 160 MB/sec. Let us further assume that another 8KByte message was received just before this one. The MCP can then spend up to 30 μ sec waiting for the LANai→Host PCI transfer (line 39) to complete. (It takes 80 μ sec for a PCI DMA assuming a bandwidth of 100MB/sec, of which at least 50 μ sec have passed during the above mentioned network receive DMA). Thus, the MCP could take as much as 110 μ sec before detecting the buffer in the send-buffer-queue. Comparing this to the end-to-end message latency of 55 μ sec, this is clearly an unacceptable latency jitter. Other threats to predictability arising from PCI bus contention also exist, and are discussed individually in the sections below.

6.2.1.2 BDM-RT

A simplified pseudo-code of the BDM-RT MCP is shown below.

```
1 while (1)
2     do
3         /* Sending messages */
4         if (An initiated network send DMA has completed) or
5             (No network send DMA was initiated)
6             then
7                 if (An initiated network send DMA has completed)
8                     then
9                         Put buffer back into free-send-buffer queue
10                    fi
11
12                /* Sending: To Network */
13                if (An initiated Host→LANai PCI DMA has completed)
14                    then
15                        Start DMAing the buffer out the network
16                    fi
17
18                /* Sending: Host→LANai */
19                if (Host has queued a buffer to send)
20                    then
21                        Initiate Host→LANai PCI DMA for the buffer.
22                        /* We are sure that there is no other ongoing
23                        LANai→Host or Host→LANai PCI DMA */
24                    fi
25                fi
26
27                /* Receiving: From network */
28                if (Message is waiting to be received from the network)
29                    then
30                        Initiate receive-DMA
31                        /* Do not block until DMA finishes */
32                    fi
33
34                /* Receiving: Buffer management */
35                if (An initiated network receive DMA has completed)
36                    then
```

```

37         Queue the buffer in the received-buffers-queue
38     fi
39
40     /* Receiving: LANai→Host */
41     if (Host asks for a message with a specified TAG)
42     then
43         Search received-buffers-queue for message with this tag
44         If found, initiate LANai→Host PCI DMA
45         /* We are sure that there is no other ongoing
46         LANai→Host or Host→LANai DMA */
47     fi
48
49     if (An initiated LANai→Host PCI DMA has completed)
50     then
51         Hand buffer to host
52     fi
53 done /* while */

```

Design Rationale: The main-loop of the BDM-RT MCP resembles that of BDM-MCP to a large extent, mainly because only those portions that affected the predictability of protocol processing and message latency were replaced from BDM-MCP. It must be noted at this point that the semantics of PCI DMA are different for BDM-RT as discussed in the subsection 5.3.4. This allows us to make an assumption that the DMA engine is not carrying out a transfer when access to the PCI bus is sought (see lines 21 and 44 in the pseudo-code). This removes two blocking statements from BDM. As timeliness is of essence to BDM-RT, the blocking network receive DMA that existed for minimal latency in BDM has been replaced by a non-blocking one.

Effect on Performance: The replacement of the blocking receive DMA from BDM (line 34 in BDM pseudo-code) increases the latency of received messages because LANai CPU activity has a negative effect on DMA bandwidth (Myricom Inc. 1996). Non-blocking receive DMAs are particularly detrimental to short-

message latencies, because the polling period (time spent in a single main-loop) is of the same order ($\approx 6-8 \mu\text{sec}$) as the receive DMA latency. Other effects of performance are examined below in association with the individual design components that cause them.

6.2.2 Message receipt

6.2.2.1 BDM

In BDM, a call to `BDM_Frame_rcv()` looks for a received message that has already been transferred to the host. The BDM MCP transfers all received messages to the host as soon as they are received. A call to `BDM_Frame_rcv()` succeeds if there is at least one message in the “shadow” received-messages-queue on the host.

Rationale: The rationale for this design was discussed in 5.2.3.1.

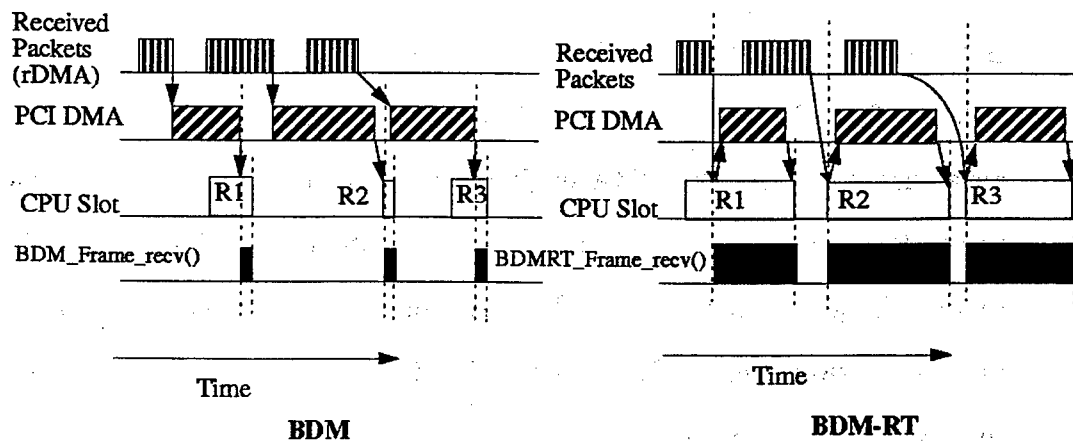


Figure 6.1: Comparison of Receive Semantics between BDM and BDM-RT

Effect on Predictability: As mentioned above, BDM-MCP transfers incoming messages to host memory on a best-effort basis. The PCI bus, as a resource, lacks

any kind of management as there is no scheduling of PCI DMA transfers. This adds to the unpredictability in message latency because of contention of the PCI bus both between Myrinet packets (outgoing vs. incoming) and between Myrinet packets and other non-real-time PCI traffic. This design can also cause undesired priority inversion. For example, let us assume the host sends a high priority packet by issuing a call to `BDM_Frame_send()`. Assume that a low-priority packet is received just before this operation and is being DMA-ed to its location in host memory. In this case the PCI bus becomes unavailable to the high-priority send-packet until the low-priority receive-packet is fully transferred to the host. This priority inversion is avoided in BDM-RT by scheduling DMA transfers during a task's allotted CPU time.

6.2.2.2 BDM-RT

The receive function `BDMRT_Frame_rcv()` involves three steps:

- The host sets a LANai flag to request the MCP for a message;
- The MCP looks in DLL (see section 5.5) for a matching message and sets another flag indicating the result. It then initiates a PCI DMA of the matched message;
- The host function returns failure if not found. If found, the host waits for DMA transfer completion.

Rationale: The rationale for this design was discussed in 5.3.3.

Effect on Performance: The three main aspects of BDM-RT receive semantics are as follows: (a) PCI DMA occurs only after a call to `BDMRT_Frame_rcv()`, (b) The MCP initiates and finalizes DMA transfers, and

(c) The `BDMRT_Frame_rcv()` function blocks the host CPU until the transfer is complete.

By delaying the LANai-to-Host PCI DMA until a call to `BDMRT_Frame_rcv()` is made, BDM-RT adopts a non-work-conserving approach. In general, this approach reduces performance because of potential non usage of the PCI bus in the presence of ready data. Message latency increases because of potential idle time spent by the received packet in LANai buffers.

Latency degradation occurs because of the host polling over a DMA completion flag that is set by the MCP. The host has to access the polled value across the PCI bus and hence interferes with the ongoing transfer by contending for the PCI bus. The LANai interface hardware architecture assigns higher access priority to the host CPU as compared to the Host-LANai DMA engine (Myricom Inc. 1996). This worsens the impact of polling on PCI DMA latency. BDM-RT has however been designed to minimize this negative effect by beginning the polling operation only after a large fraction (say 90 percent) of the average DMA duration has elapsed. Also, once polling begins, the completion flag is examined not more than once in 5 μ sec, to limit the number of accesses to the PCI bus. This decreases the average number of accesses to about half the number in a tightly polled loop (average read time is 2.5 μ sec).

The major performance degradation arises because of "blocking DMA." Performing blocking DMA greatly degrades bandwidth, by decreasing the amount of message-pipelining. Processor utilization is also reduced because useful computation cannot be overlapped with an ongoing DMA transfer. In the presence of large amounts of contending PCI bus traffic from non-real-time Linux processes, blocking DMA promises lower average latency than regular DMA by minimizing

the effect of contention. However, in the absence of severe contention, a latency degradation rather than enhancement is incurred because of the abovementioned three-step receipt procedure.

6.2.3 Sender-side host to LANai transfer

6.2.3.1 BDM

BDM uses PCI DMA initiated by the MCP for all Host-to-LANai DMA transfers. Detection of DMA completion is also done by the MCP. When a call to `BDM.Frame_send()` is made, the buffer is put into the ready-to-send queue and the function returns immediately. In its main-loop, BDM-MCP detects this buffer and initiates a PCI DMA to transfer the data to LANai buffers before sending it out the network. The rationale for this design was presented in the BDM Design section in 5.2.4.

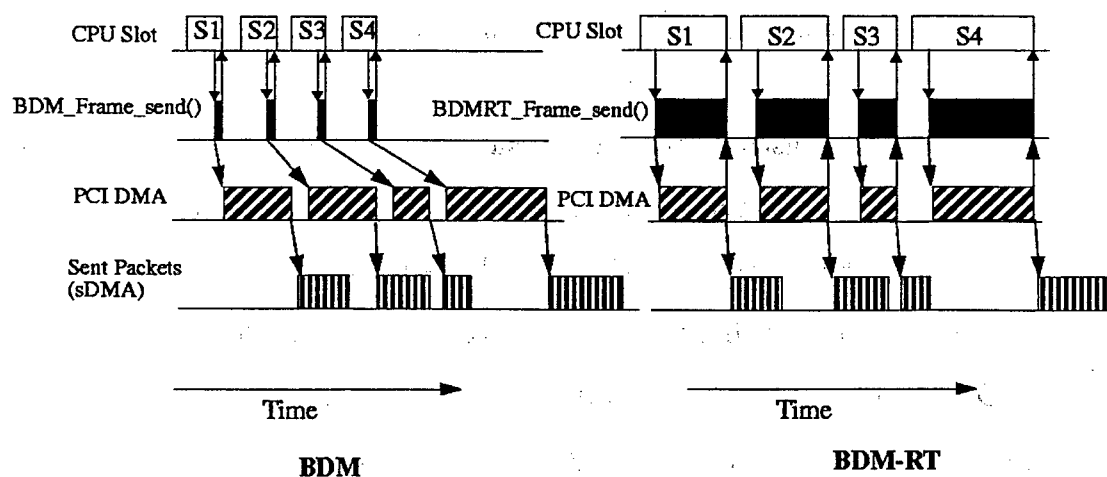


Figure 6.2: Comparison of Send Semantics between BDM and BDM-RT

Effect on Predictability: This design is not suitable for predictable data transfer between the host and LANai memory for reasons cited in the sub-section 5.2.4.

Potential contention of PCI bus usage can occur from other best effort non-BDM Linux processes that may be scheduled on the CPU during the PCI DMA transfer.

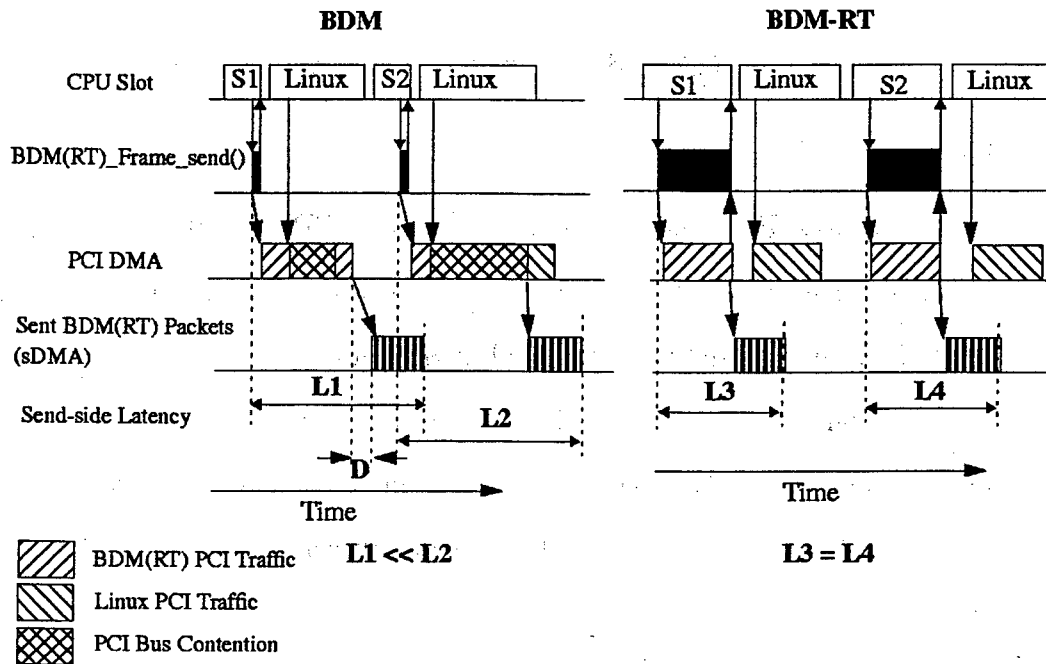


Figure 6.3: BDM Send Semantics: Contention with Linux for PCI bus

Figure 6.3 illustrates the effect of such non-real-time PCI traffic on BDM traffic PCI DMA latency. The figure shows two messages of equal length sent by two tasks S1 and S2. For BDM we notice that the latencies L1 and L2 are higher and less predictable than their BDM counterparts L3 and L4. The unpredictability in the BDM-MCP main-loop discussed in 6.2.1.1 can aggravate the problem by adding more uncertainty to the actual start time of the PCI DMA. This is shown by the duration D for BDM.

6.2.3.2 BDM-RT

BDM-RT uses “blocking DMA” semantics which causes `BDMRT_Frame_send()` to hold the CPU for an empirically calculated average duration of PCI DMA. The rationale was presented in the BDM-RT design section under 5.3.4

Effect on Performance: The main effect of “blocking DMA” is bandwidth reduction because of wasted CPU time during the DMA transfer. For high-performance applications, blocking communication calls reduce their ability to overlap useful computation with communication. In the presence of heavy extraneous non-real-time PCI traffic, the overall average latency actually improves because of a significantly lower jitter component.

6.2.4 Header information

6.2.4.1 BDM

The BDM packet header is described in Henley et al. (1997). Prominent header fields are the route-information, message length, and message protocol type (level of reliability). See Henley et al. (1997) for more information on BDM header fields.

6.2.4.2 BDM-RT

The BDM-RT packet header consists of a tag field in addition to those present in a BDM packet header.

Rationale: The tag field enables BDM-RT to demultiplex received messages among various receiving tasks. The tag field was added to BDM-RT to avoid priority inversion of received messages by allowing a high-priority receive to precede a low-priority one even though the latter packet arrives at the LANai first. This

also allows easier building of MPI/RT channels by using one tag to represent a channel throughout an MPI/RT program.

Effect on performance: In theory, the latency of BDM-RT increases because of the increased length of the header. However, the effect of an extra tag field (1 byte) in the header is negligible compared to the actual latency of zero-byte messages ($\approx 50 \mu\text{sec}$).

6.2.5 Sender-Side message delivery order

Both BDM and BDM-RT send messages in First-In-First-Out (FIFO) order. At the receiver's end, `BDM_Frame_recv()` returns messages in the order of their receipt at the LANai, while `BDMRT_Frame_recv()` returns the first received message with a matching tag.

6.2.5.1 BDM

Rationale: FIFO queue processing generally involves the least queue processing overhead. Further, these queues are implemented as single-producer single-consumer queues with the host library and BDM-MCP playing the roles of producer and consumer and require no overhead for mutual exclusion.

Effect on Predictability: FIFO processing is inherently QoS-insensitive. However, if a higher layer is capable of QoS-based prioritizing of messages before handing them off to the MCP, QoS-sensitivity can be introduced. At the receiving end also, restricting applications to receive messages in their order of arrival at the LANai causes undesirable priority inversion. Building a de-multiplexing layer above BDM does not alleviate the problem because priority inversion occurs in

protocol processing, in our case - the LANai-to-host PCI DMA - done *inside* the BDM layer.

6.2.5.2 BDM-RT

Rationale: At first glance, FIFO queuing appears inadequate for BDM-RT to function as a QoS-sensitive messaging layer, and to avoid priority inversion (Mehra, Indiresan, and Shin 1996) (Lee et al. 1996). However, on closer examination FIFO delivery is seen to suffice based on the following reasoning: BDM-RT has been designed to efficiently layer time-based MPI/RT channels. The MPI/RT Commit() function (Kanevsky, Skjellum, and Watts 1997) will map each channel into a PromisQoS RT task and TURTLE scheduler will order the execution of send-side channels in a manner consistent with the channel's QoS requirements. In other words, channel protocol processing including host-to-LANai DMA on the send side will be scheduled in the order of earliest delivery deadlines. The packets then undergo minimal processing at the MCP before being sent out in FIFO order. Thus messages received from a single node are always in prioritized order. However, messages received from different source nodes can still suffer priority inversion if the receive function can retrieve messages only in FIFO fashion. To solve this problem, BDM-RT provides for a tag-based receive mechanism. As long as a message has been received at a node's LANai buffers, it can be received immaterial of its position in the receive queue. Best-effort tasks do not have explicit support in this version of BDM-RT. In the presence of best-effort tasks using BDM-RT, one FIFO queue will not suffice. One needs at least two separate queues in order to mask interference from best-effort tasks and still continue to meet the deadlines of

real-time tasks. At the same time, the adopted queue processing strategy should be fair to best-effort traffic and not starve it of link or CPU bandwidth.

Effect on Performance: At the sender's side, there is no performance trade-off compared to BDM because of the identical FIFO queuing of messages. Queue processing complexity is $\Theta(1)$. At the receiver's side, the tag-based de-multiplexing mechanism involves searching, adding, and deleting nodes in the linked list (DLL) of received messages, which is $\Theta(n)$. Alternately, if the linked list was organized as a binary search tree, the order of complexity of queue manipulation would be $\Theta(\log_2 n)$. In our case, at most 8 buffers can be queued at the receiver's end. For $N=8$, the gain from a binary tree is insignificant, specially compared to the overall latency ($\approx 50 \mu\text{sec}$) which is nearly two orders of magnitude greater than worst-case linked list manipulation overhead ($\approx 1 \mu\text{sec}$, based on measured LANai instruction execution speeds).

6.3 Summary

Based on the analysis of BDM and BDM-RT design, we see that nearly all components of BDM that optimize its performance are unsuitable for real-time communication. The converse is also true - components of BDM-RT that enhance its predictability involve performance compromises. In particular, the BDM-MCP main-loop performs certain blocking operations that decrease the predictability of protocol processing delays. BDM lacks explicit management of the PCI bus and the Myrinet switch, leading to resource contention. Undesirable priority inversion exists due to the FIFO receipt scheme. On the other hand BDM-RT achieves predictability at the cost of performance.

CHAPTER VII

EXPERIMENTS, RESULTS, AND ANALYSIS

This chapter presents the experimental methodology and the results obtained from these experiments. The aim of these experiments is to demonstrate the validity of the hypothesis and to corroborate the analysis of design for BDM and BDM-RT presented in chapter VI. The experiments chiefly compare the timeliness and performance of BDM and BDM-RT for various message sizes and message-passing scenarios. Most results are based on timing measurements using the global clock described in 2.2.1.

7.1 Experiment setup

This section describes the configuration in which experiments were performed. It also points out the limitations, and estimates the accuracy, and overhead involved in gathering performance metrics.

7.1.1 Hardware and software

All experiments were performed on a two-node Pentium Pro Myrinet cluster. The host and network hardware specifications were also listed in 2.1. Software specifications were listed in 2.1.1. The main reasons for limiting the experiment platform to a two node, one Myrinet switch is the current state of development of PromisQoS, and the global clock.

7.1.2 Timers used

The clock synchronization module provides a 64-bit nano-second resolution virtual clock on all slave nodes (see 2.2.1). This clock is based on the 64-bit nano-second resolution APIC Time Stamp Counter (found on all Pentium chip-sets) of the master node. Test programs use global time-stamps to compute end-to-end metrics such as latency, bandwidth, and latency-jitters.

Internal metrics such as PCI DMA latencies, PCI DMA jitters, and MCP processing overhead use the on-board LANai RTC (Myricom Inc. 1996) with a resolution of 0.5 μ s. More resolution is desirable in future LANai hardware.

7.1.3 LANai storage limitation

Metrics such as PCI DMA latency and MCP processing overhead cannot be gathered for long durations because of limited storage space on LANai SRAM. For example, on a 1 MB board, about 400 KB of free space is available for metrics. Assuming that each metric includes a 32-bit RTC time-stamp and a 32-bit integer metric value (totaling to 8 bytes per set), we can record a maximum of 50,000 sets. Assuming a task with period 500 μ sec, this limits the recording duration to 25 seconds. The simplest way to overcome this limitation without periodically transferring out metric data from LANai memory (which can potentially interfere with the RT traffic) is to record only a subset of events. For instance, in the measurement of LANai-to-Host DMA latencies, latency is recorded only if it exceeds a certain value, i.e the upper bound on latency for that message length. This greatly reduces the amount of data that needs to be stored, specially if the system meets its latency bounds most of the time.

7.1.4 Metric overhead

Both the host and the LANai introduce processing overhead associated with instrumentation for gathering performance metrics by both the host and the MCP. The host gathers metrics on the number of PCI DMA transfers that exceeded their latency bounds and the total number of PCI DMA transfers performed by a program. Test programs also record global time-stamps on the host, for latency and bandwidth calculation. Given the high speed of the Pentium processor (200MHz), and low access times for the APIC TSC (30-50 ns) the instrumentation overhead is negligible compared to the overall host protocol processing time. The MCP gathers metrics such as individual PCI DMA latencies, polling duration of the MCP main-loop, and individual network DMA latencies and stores them in statically allocated LANai memory. A typical metric recording operation consists of recording a time-stamp and a metric value at key points in the main-loops such as initiation and completion of DMA transfers. The instrumentation overhead was estimated by empirically measuring access times for the RTC and LANai memory. It was found experimentally that the MCP spends between 3 and 5 percent of its time in a main-loop in the instrumentation code.

7.1.5 Accuracy and error bounds

As pointed out earlier, all end-to-end time measurements are done using the global clock. The global clock itself has an error bound of $\pm 4 \mu\text{sec}$ in the worst case. For latency of long messages (≥ 500 bytes), this amounts to less than 5 percent error. For short messages, the error factor is rather high, but its effect is reduced when averaged out over a long duration because of the typical "swinging" behavior of clock synchronization error around both sides of the x-axis (see figure

2.5). Individual error bounds for each performance metric are discussed in their respective sections.

For PCI DMA statistics, the error bound on DMA latency is largely governed by the time that elapsed between the actual completion of a DMA and its detection by the MCP. The MCP main-loop duration has an upper bound of 10-12 μ sec. This error is unacceptably high, specially because our interest is in measuring DMA latency jitters rather than latencies themselves. To reduce polling error, the MCP checks for DMA completion at several places in the main-loop instead of just once. Using such a scheme, the measurement error is reduced to 3-4 μ sec in the worst case (but at the same time adds to instrumentation overhead).

7.2 Latency measurements

Latency is measured by recording the global time-stamp at the sender's node just before calling `BDM(RT)_Frame_send()` and at the receiver's node just after `BDM(RT)_Frame_rcv()` returns. One-way latency is then computed by subtracting the sender's time-stamp from the receiver's time-stamp. The set-up used for this experiment is as follows: An RT task S (sender) runs on node A while an RT task R (receiver) runs on node B, both with a period of 1 ms. Both processes specify a deadline value that is marginally greater than the compute time, thus forcing the scheduler to allot CPU time exactly between their specified start-time and deadline, every period. Additionally, the sender's start-time is staggered by a delay of 50 μ sec with respect to the receiver. This is done to ensure that messages never arrive at node B before R is scheduled. Such a situation would induce error into latency measurements. The latency was averaged over 1000 messages.

7.2.1 Expected results

Both BDM and BDM-RT are expected to have comparable but relatively large zero-byte latency because of PCI DMA overhead. BDM-RT is expected to have higher latency. This is mainly because of the additional overhead in BDM-RT_Frame_recv() owed to (a) the tag-based receive mechanism and (b) performing the receive side PCI DMA only after a call to BDM-RT_Frame_recv() is issued. Tag based de-multiplexing involves the host writing tag information into LANai memory, setting a flag, and examining the flag for status of the received message. All three of these operations are performed by the host across the PCI bus (see section 6.2.2). There is extra overhead in the MCP also because of link-list traversal to identify the tagged message. In BDM, we are able to reduce this overhead to just one access across the PCI bus to check if at least one message has been fully received into host memory. However, this occurs at the cost of (a) having no tag-based de-multiplexing at the receiver's node, and (b) higher latency jitter because of the absence PCI traffic scheduling.

A second cause of latency increase is because of the difference in the network receive DMA (rDMA) semantics between BDM and BDM-RT. For best latency, when a message arrives BDM-MCP DMA's the entire message and spins until DMA completion. In BDM-RT, the MCP initiates the DMA and goes off into its main-loop to service other potential data transfers. The effect of this non-blocking receive DMA in BDM-RT is specially high on very short messages as the polling interval for DMA completion (4 to 6 μ sec) becomes comparable to DMA latencies.

7.2.2 Actual results

Figure 7.1 compares the message latency for short messages (length ≤ 512 bytes). BDM has a zero byte latency of 32 μsec , while the same is as high as 54 μsec for BDM-RT. This latency overhead in BDM-RT was explained above.

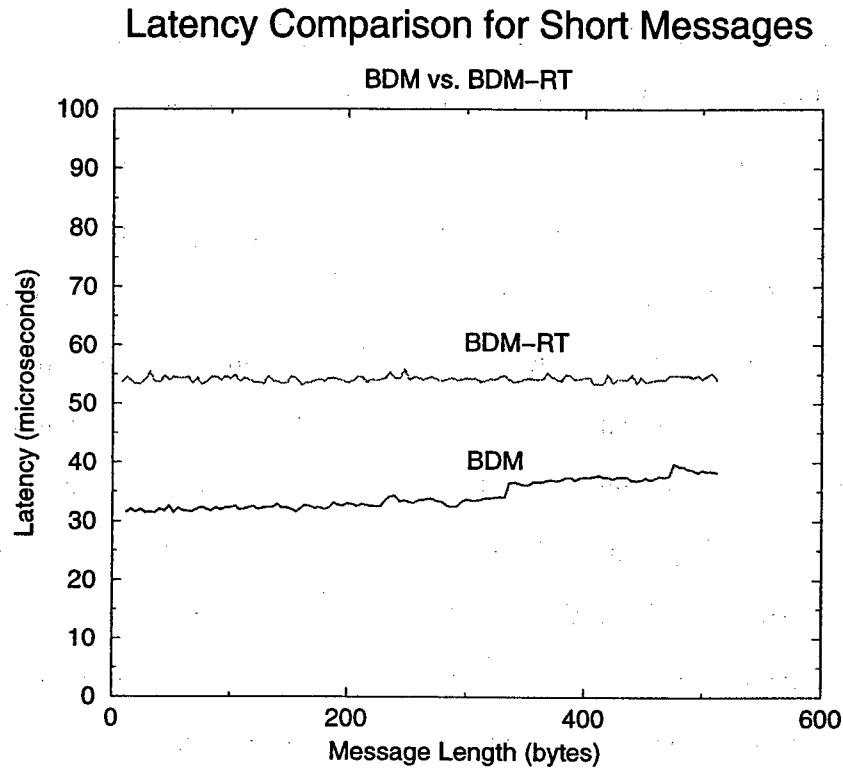


Figure 7.1: Latency for short messages

Another observation in the latency graphs is that the latency of BDM messages rises with increase in message size, whereas it remains more or less constant in BDM-RT. The increasing behavior seen in the case of BDM is typical of latency curves, and needs no explanation. The reason for BDM-RT's flat graph is two-fold. Firstly, BDM-RT is a real-time messaging layer, so it always budgets no less than the worst case latency of messages rounded off to the next higher multiple

of 5 μsec . Secondly, the receive function minimizes the negative effect of polling for DMA completion by polling in steps of 5 μsec (described in section 6.2.2.2). Because of the above reasons, the range of latencies for messages between 0-512 bytes maps to a single latency.

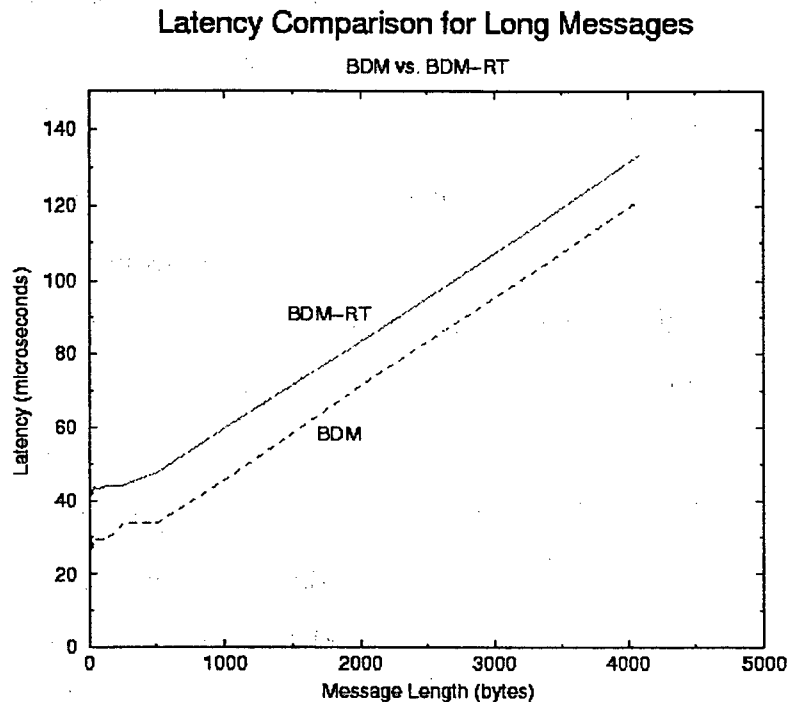


Figure 7.2: Latency for long messages

Figure 7.2 compares the latencies of messages for a longer range of message lengths. Because both BDM and BDM-RT use PCI DMA for transfers between host and LANai memory, their latency graphs look nearly identical over the range of message lengths. BDM-RT has an extra latency of about 10 μsec , because of the same overheads described in the previous subsection.

7.3 Bandwidth measurements

Bandwidth is measured by streaming messages from one node and receiving it at the other in a tight loop. The set-up used for this experiment is as follows: An RT task S (sender) runs on node A while an RT task R (receiver) runs on node B, both with a period as large as 10 ms. The period is limited to 10ms to avoid CPU contention from the Linux task, which is guaranteed 1ms of every 10ms. Task S specifies a CPU Time of 4ms, and task R specifies 4.2 ms to account for message latency. Both processes specify a deadline value that is marginally greater than their compute time, thus forcing the scheduler to allot CPU time exactly between their specified start-time and deadline, every period. Both tasks ask for a large chunk of CPU time - approximately 5 ms every period. During the allotted CPU time, the sender sends messages in a tight loop consisting of the function pair `BDM(RT)_Frame_malloc()` and `BDM(RT)_Frame_send()`. Meanwhile, the receiver receives messages in a tight loop consisting of calls to the function pair `BDM(RT)_Frame_recv()` and `BDM(RT)_Frame_free()`. The BDM unreliable protocol (Henley et al. 1997) is used for bandwidth measurement because MPI/RT Channels will be implemented using this protocol. Using this protocol in a tight sending loop can cause receiver-buffer overflows if the receiver is slower than the sender, leading to message loss. To get an accurate measure of bandwidth, message loss should be avoided by tuning the sender's rate to that of the receiver. This involves finding the difference in sender and receiver rates, and adding a suitable duration of inactivity in the sender's loop. Tuning the senders rate exactly to the receiver's rate is iterative and rather time-consuming to achieve practically. As an alternative, unreliable bandwidth was measured as the receiver's rate in

the presence of lost messages which do not count for bandwidth calculation. The measured bandwidth was averaged over 1000 messages.

7.3.1 Expected results

BDM is expected to have significantly better bandwidth than BDM-RT because of “blocking” PCI DMA transfers in BDM-RT. In BDM, the `BDM_Frame_send()` call is non-blocking thus allowing the user to queue up PCI DMA transfers while one is in progress. In BDM-RT, `BDMRT_Frame_send()` returns only after the holding on to the CPU until the estimated PCI DMA latency elapses. At the receiver’s side, `BDM_Frame_recv()` returns a pointer to a message that has already been transferred to host memory, while `BDMRT_Frame_recv()` initiates and waits on the completion of DMA between LANai and host memory.

7.3.2 Actual results

Figure 7.3 shows the bandwidth of BDM and BDM-RT. The theoretical PCI bandwidth for 32-bit, 33MHz PCI is 132 MBytes/sec. BDM achieves about 88 percent of this bandwidth at 7800 bytes. As expected, BDM-RT has a lower bandwidth owed to the “blocking” PCI DMA.

7.4 Latency jitter

Quality of Service for MPI/RT Channels consists of providing guaranteed bandwidth and a latency bound for channel messages. In this experiment we measure end-to-end message latency jitter for BDM and BDM-RT. The latency program described in section 7.2 is used for this experiment as well. Disk activity is shut off during this experiment to avoid PCI bus contention from the disk driver.

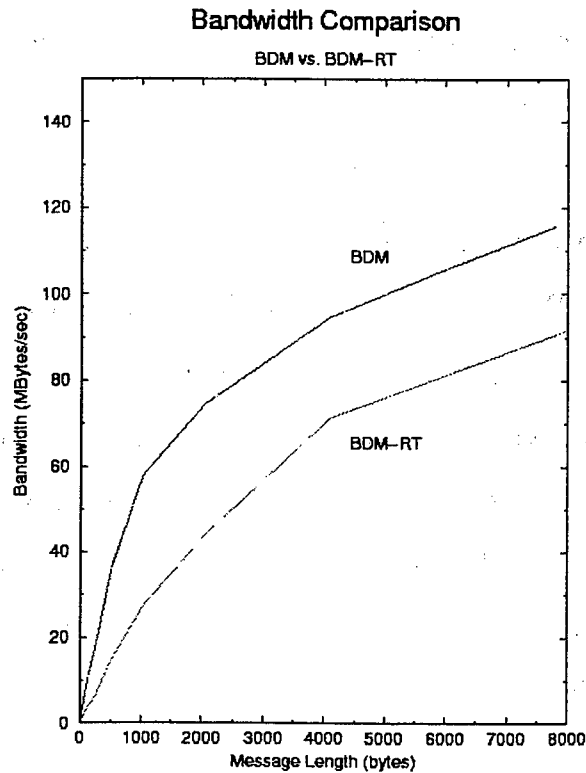


Figure 7.3: Bandwidth

7.4.1 Expected results

The latency jitter for BDM is expected to be significantly larger than that for BDM-RT. The chief source of latency jitter applicable to this experiment is PCI bus contention from non-BDM(RT) processes. BDM-RT is designed to minimize the effect of PCI bus contention, by disallowing extraneous PCI transfer initiations during a BDM-RT PCI transfer (see Chapter 6). In the absence of disk activity, BDM-RT is expected to provide a hard bound on the latency jitter, while BDM cannot.

7.4.2 Actual results

Figures 7.4 and 7.5 show the end-to-end latencies of BDM and BDM-RT for 4KB sized messages. These figures also show the latency incurred in two intermediate stages of message transfer: the host-to-LANai and LANai-to-host PCI DMA latencies. For BDM, it is clear that the unpredictability in host-to-LANai DMA latencies. For BDM, it is clear that the unpredictability in host-to-LANai DMA latency is the chief factor influencing the overall latency jitter. This jitter component is nearly negligible in the case of BDM-RT. BDM-RT can be seen to be better suited for QoS provision than BDM.

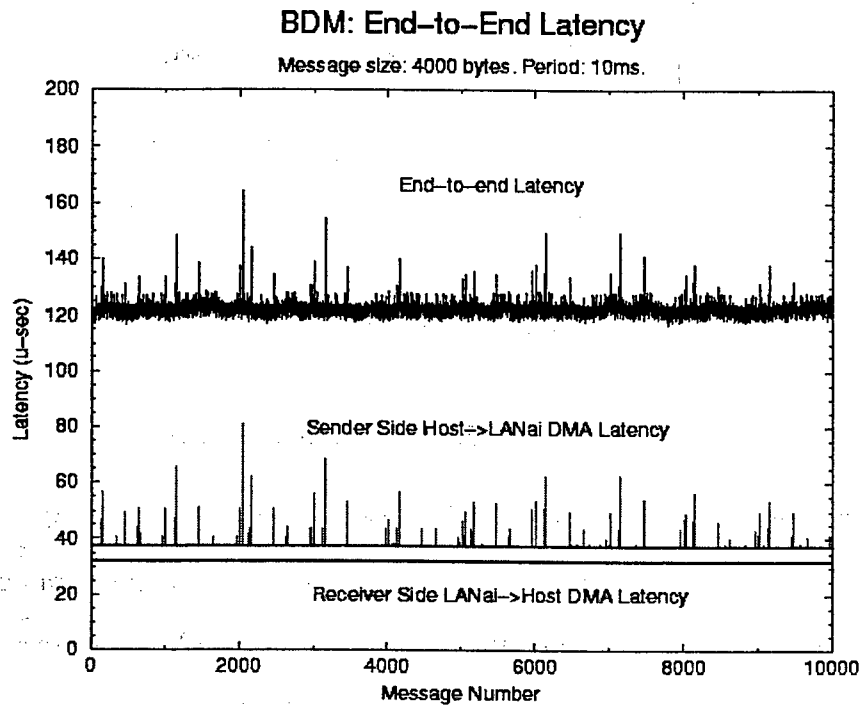


Figure 7.4: BDM: Latency Jitter

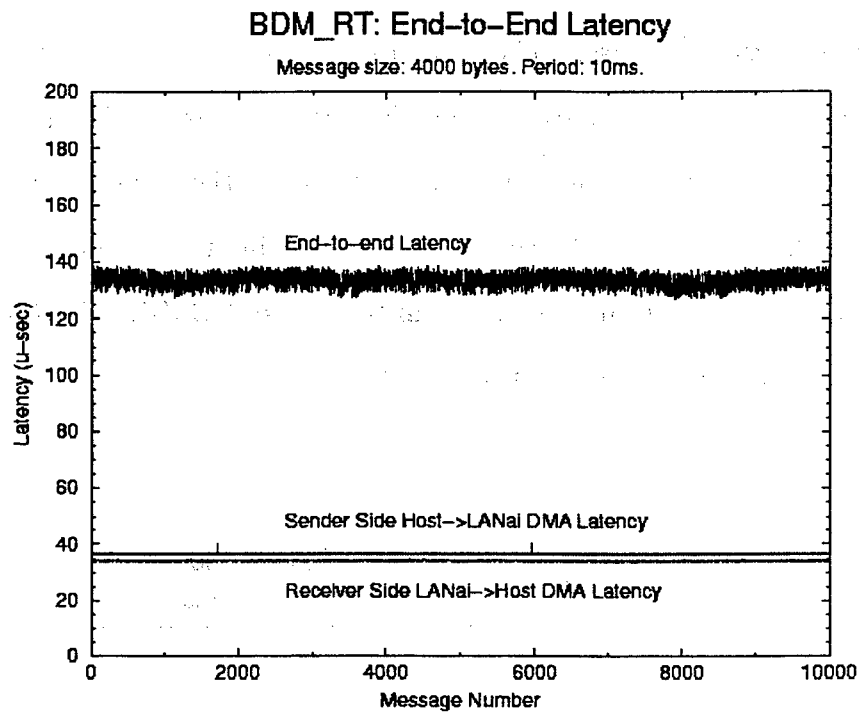


Figure 7.5: BDM-RT: Latency Jitter

7.5 Effect of external traffic

The effect of other non-BDM(RT) tasks contending with BDM(RT) tasks for the PCI bus is measured by introducing synthetic ethernet traffic. The latency program is used for this experiment as well. The sending and receiving nodes are ping-ed by a third node with 8000 byte packets every 1 second.

7.5.1 Expected results

We expect to see clear spikes in PCI DMA latency for BDM, and minimal or no effect for BDM-RT, for the reasons cited in the previous section.

7.5.2 Actual results

Figure 7.6 shows the result of artificially induced PCI contention on BDM and BDM-RT message latencies. As expected, BDM suffers latency jitters while BDM-RT is nearly unaffected.

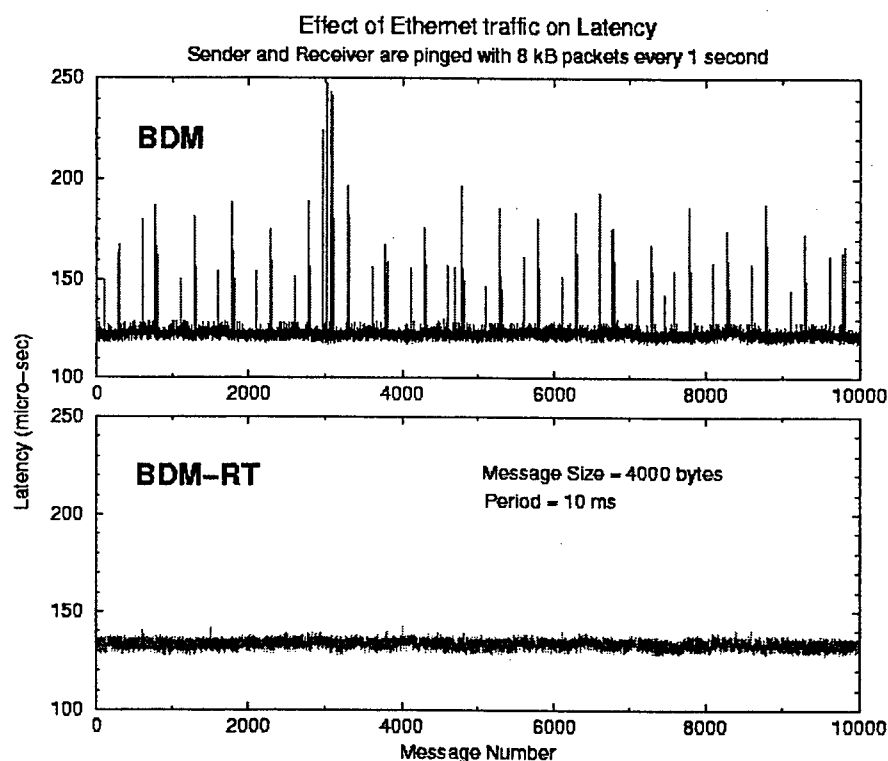


Figure 7.6: Effect of non-real-time ethernet traffic

7.6 Summary of results

All experiments yielded expected results and confirm the hypothesis. The QoS experiments showed that BDM-RT outperforms BDM in terms of latency jitter and immunity to external PCI traffic. BDM is seen to be unsuitable for real-time communication because of its inability to meet the fundamental requirement of predictable message passing. Experiments on performance metrics showed

that BDM-RT has lower bandwidth and higher latency compared to BDM. In summary, BDM is unsuitable for real-time communication, and that BDM-RT requires performance tradeoffs to achieve predictability. These results corroborate the analysis of the design differences between BDM and BDM-RT from the previous chapter.

CHAPTER VIII

CONCLUSIONS

This thesis hypothesized that a fundamental dichotomy exists between the design of low-level real-time and high-performance messaging layers. The hypothesis was scoped over the design of a real-time messaging layer with performance requirements, over Myrinet clusters of PCs. This thesis is applicable in general to networks of COTS computers characterized by high-speed communication, low-latency cut-through switching, and a programmable network interface.

The motivation for this work was the requirement for a real-time messaging layer with reasonably high performance for the development of a time-based Channel implementation of the MPI/RT middleware with QoS support. Myrinet was chosen as the network because of its high speed and low bit-error rates. The absence of time-based real-time messaging layers on Myrinet and limited past research on worm-hole routed networks provided the motivation to design and implement BDM-RT - a real-time messaging layer.

The basic requirements of real-time communication were reviewed in chapter 3. The goals and design choices for high-performance messaging layers on Myrinet were presented in chapter 4. It was seen that BDM shares common design goals with other popular Myrinet messaging layers such as FM, GM, BIP, U-Net, and VMMC. It was also noted that BDM and all other cited high-performance

messaging layers contained few or no mechanisms to address the requirements of real-time communication previously presented in chapter 3.

Chapter 5 presented the detailed design of BDM and BDM-RT and discussed the rationale behind their design. BDM-RT is the first known hard real-time messaging layer on Myrinet. BDM-RT can provide bounded end-to-end latency by adopting several techniques that improve its predictability. BDM-RT isolates the shared PCI bus from best-effort traffic by introducing the concept of “blocking PCI DMA”. BDM-RT supports high-quality transmission of clock messages to facilitate the implementation of a fine grain ($\pm 4 \mu\text{sec}$) global clock with acceptably low synchronization overheads. The BDM-RT MCP offers bounded response time and bounded protocol-processing time. Combined with the fine grain clock, this ensures conflict-free end-to-end message transfer.

Compared to FM-QoS, the only other known implementation of a QoS based messaging layer on Myrinet, BDM-RT was shown to be superior in the following aspects. BDM-RT can provide end-to-end guarantees rather than just LANai-to-LANai guarantees provided by FM-QoS. The overhead for BDM-RT clock synchronization is an order of magnitude less than for the self-synchronizing schedules of FM-QoS. The BDM-RT clock synchronization algorithm also appears to scale better for multiple-switch networks.

The validity of the hypothesis was demonstrated in chapter 6 by first pointing out that performance-predictability trade-offs are inevitable, given a set of hardware architectural constraints. The design of various components of BDM and BDM-RT were compared and contrasted. BDM's chosen design was shown to have an undesirable effect on its predictability. BDM lacks provisions for explicit management of resources such as the PCI bus and Myrinet switch. Traffic

isolation is absent in BDM, and protocol processing is neither time-bound nor free of undesired priority inversion. Similarly, the goal of predictability in BDM-RT was shown to impact its performance negatively. Mechanisms in BDM-RT such as “blocking PCI DMA” for resource isolation, bounded-time protocol processing in the MCP, and tag-based receiving to avoid priority inversion were shown to lower its performance.

This analysis was corroborated in Chapter 7 with experimental verification of performance and QoS parameters of BDM and BDM-RT. The actual results from these experiments matched closely with the expected results. BDM-RT shows significantly better ability to meet QoS requirements by providing bounded jitter on latency and isolation from external non-real-time ethernet traffic. Additionally, BDM-RT was seen to have lower bandwidth and higher latency as compared to BDM.

Based on the presented analysis and experimental verification with BDM and BDM-RT, and given the degree of similarity in the design and goals of various high performance messaging layers such as FM, GM, AM, and BDM, we conclude that the hypothesis has been verified to be true.

8.1 Future work

It would be interesting to study the applicability of this thesis work to other high-speed networking technologies such as GigaNet (GigaNet, Inc. 2000), ServerNet (Horst and Garcia 1997), and ATM. In networks such as GigaNet and ServerNet processing at the network interface is implemented in hardware or firmware. On Myrinet, BDM-RT implements predictable end-to-end communication by providing bounded response time software (MCP) at the

network interface. This ensures that link traffic scheduled by a global clock at the host will not encounter resource conflicts at the network medium or at the network switches. A study of the predictability of protocol processing at the network interface and the predictability of switch fall-through times for other high-speed networking technologies is the first step towards applying this thesis to them.

REFERENCES

- Apte, M., S. Chakravarthi, A. Pillai, A. Skjellum, and X. Zan. 1999. Time-based Linux for real-time NOWs and MPI/RT. In *Proceedings of the Real-Time Systems Symposium*, pages 221-222, Los Alamitos, CA. IEEE Computer Society.
- Barbanov, M. and V. Yodaiken. 1997. Introducing real-time linux. *Linux Journal* (February):19-23.
- Bhoedjang, R. A., T. Ruhl, and H. E. Bal. 1998. User-level network interface protocols. *IEEE Computer* 31 (November):53-60.
- Boden, N. J., D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. 1995. A Gigabit-per-second Local Area Network. *IEEE-Micro* 15 (February):29-36.
- Cilingiroglu, A., S. Lee, and A. Agrawala. 1997. *Real-time communication*. Technical Report CS-TR-3740, University of Maryland, College Park.
- Connelly, K. and A. Chien. 1997. FM-QoS: Real-time communication using self-synchronizing schedules. In *Proceedings of Supercomputing*.
- Dubnicki, C., A. Bilas, K. Li, and J. Philbin. 1997. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the International Parallel Processing Symposium*, pages 388-396.
- Games, R., A. Kanevsky, P. Krupp, and L. Monk. 1995. Realtime communication scheduling for massively parallel processors. In *Real-Time Technology and Applications Symposium*, pages 76-85. IEEE.
- GigaNet, Inc. 2000. GigaNet. <http://www.giga-net.com> Last Accessed: March 19, 2000.
- Henley, G., N. Doss, T. McMahon, and A. Skjellum. 1997. *BDM: A multiprotocol Myrinet Control Program and host application programmer interface*. Technical Report MSSU-EIRS-ERC-97-3, Mississippi State University.
- Horst, R. W. and D. Garcia. 1997. ServerNet SAN I/O architecture. In *Proceedings of Hot Interconnects V*.
- Indiresan, A., A. Mehra, and K. G. Shin. 1995. *Design tradeoffs in implementing real-time channels on bus-based multiprocessor hosts*. Technical Report CSE-TR-238-95, University of Michigan.

- Kanevsky, A., A. Skjellum, and J. Watts. 1997. Standardization of a communication middleware for high-performance real-time systems. In *Proceedings of the Real-Time Systems Symposium*.
- Lee, C., K. Yoshida, C. Mercer, and R. Rajkumar. 1996. Predictable communication protocol processing in real-time Mach. In *Proceedings of 2nd Real-Time Tech., and Appl. Symposium*.
- Mehra, A., A. Indiresan, and K. G. Shin. 1995. *Resource management for real-time communication: Making theory meet practice*. Technical Report CSE-TR-281-96, University of Michigan.
- Mehra, A., A. Indiresan, and K. G. Shin. 1996. Resource management for real-time communication: Making theory meet practice. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 130-138.
- Mehra, A., A. Indiresan, and K. G. Shin. 1996. Structuring communication software for quality of service guarantees. In *Proceedings of the Real-Time Systems Symposium*, pages 144-154.
- MPI Software Technology, Inc. 2000. BDM/Pro. <http://www.mpi-softtech.com> Last Accessed: March 19, 2000.
- Mryicom, Inc. 1998. GM. <http://www.myri.com/GM/doc/gm.toc.html> Last Accessed: July 20, 1999.
- Myricom. 1996. Lanai 4.x specification. <http://www.myri.com/scs/documentation/mug/development/LANai4.X.doc.txt> Last Accessed: Jan 30, 2000.
- Pakin, S., M. Lauria, and A. Chien. 1995. Illinois fast messages (FM) for Myrinet. In *Supercomputing*.
- Prylli, L. 1997. *BIP user reference manual*. Technical Report TR97-02, LIP/ENS-LYON, University of Lyon.
- Shanley, T. and D. Anderson. 1995. *PCI system architecture*. Addison Wesley Publishing Company, Menlo Park, CA.
- Snir, M., S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. 1995. *MPI - The complete reference*. The MIT Press, Cambridge, MA.
- Solari, E. and G. Willse. 1998. *PCI hardware and software architecture and design*. Annabooks, San Diego, CA.
- Stankovic, J. A. and K. Ramamritham. 1990. What is predictability for real-time systems? *Real-Time Systems* 2:247-254.
- Stankovic, J. A. and K. Ramamritham. 1993. *Advances in real-time systems*. IEEE Computer Society Press, Los Alamos, CA.
- Stevens, R. 1994. *TCP/IP illustrated, volume 1: The protocols*. Menlo Park, CA.

- Sunderam, V. S., G. A. Geist, J. Dongarra, and R. Manchek. 1994. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing* 20 (April):531-545.
- Tezuka, H., F. O'Carroll, A. Hori, and Y. Ishikawa. 1998. Pin-down cache: A virtual memory management technique for zero-copy communication. In *12th Int. Parallel Processing Symposium*, pages 308-314.
- von Eicken, T., A. Basu, V. Buch, and W. Voegls. 1995. U-net: A user level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 303-316.
- von Eicken, T., D. E. Culler, S. C. Goldstein, and K. E. Schauer. 1992. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th ACM Symposium on Computer Architecture*, pages 256-266.
- Zhang, L. Y., J. W. S. Liu, Z. Deng, and I. Philip. 1999. Hierarchical scheduling of periodic messages in open system. In *Proceedings of the Real-Time Systems Symposium*, pages 350-359.

APPENDIX A
HARDWARE MECHANISMS FOR PREDICTABILITY

Below are some hardware and firmware solutions to achieve predictable PCI latency and bandwidth in the presence of bus contention from other non-real-time traffic.

A.1 PCI register initialization

The Peripheral Component Interconnect (PCI) 2.0 Standard (Solari and Willse 1998) provides a set of configuration registers that are either initialized by configuration software at boot-up time, or have a hardwired value. Some of the PCI configuration registers can be used to mitigate the effects of bus contention on the PCI bus access time and PCI DMA latency for Myrinet traffic as described below. As a background, PCI registers of significance to this discussion are listed below.

Min_Gnt Min_Gnt is a read-only register applicable only to bus master devices.

A non-zero value indicates, in multiples of 250ns, how long the master would like to retain PCI bus ownership every transaction, for best performance. A zero value indicates that the device has no such requirement on ownership time-slice.

Max_Lat Max_Lat is a read-only register that specifies how often, in multiples of 250ns, a device needs to gain access to the PCI bus.

Master Latency Timer The Master Latency Timer (MLT) prevents bus masters from monopolizing the bus. The MLT value defines the minimum amount of time, in PCI clock ticks, the bus master is allowed to retain ownership of the bus once a transaction starts. The MLT value is permitted to be hardwired only if the bus master is incapable of performing more than two data phases per transaction (Solari and Willse 1998). Otherwise, MLT

should be a read/write register. Max_Lat and Min_Gnt values are normally used by boot-up software to determine the MLT value for each device.

Assigning suitable values for MLT can improve the real-time behavior of Myrinet traffic at the cost of sacrificing optimal performance of other devices. For instance, by using a high value of MLT for itself, the Myrinet device can prevent losing ownership of the bus for that many PCI clock ticks. In combination with this, a MLT low value for all other PCI devices will reduce bus acquisition latency for the Myrinet device. The feasibility of the above solution is affected by the presence of devices in the real world that use a hardwired value for their MLT register, although this violates the PCI Standard. This makes it impossible to change the MLT register value for such devices.

A.2 Customizing PCI arbitration

The PCI 2.0 Standard does not define the arbitration logic to arbitrate access requests from different bus masters sharing a PCI bus. It is up to the PCI chip-set manufacturer to implement any fair arbitration scheme that does not cause a deadlock. The PCI arbiter implements a simple round-robin scheme in most commercial chip-sets. Consequently, a solution to improve predictability of Myrinet DMA transfers is to use priority-based arbitration, using highest priority for the Myrinet DMA bus-master. This option was not chosen because it demands specialized hardware, i.e. a customized PCI arbiter or a PCI chip-set with a programmable arbiter. This conflicts with one of the goals of PromisQoS - that of being based on a COTS platform.

A.3 Dual PCI bus

A possible solution to avoiding bus contention between real-time and non-real-time traffic is to use a dual-PCI bus configuration, with a dedicated bus for real-time Myrinet traffic. This is acclaimed by researchers in real-time systems as the most effective and simple hardware solution in the presence of contention from non real-time traffic. This option was also discarded because commercial desk-tops do not have such a configuration.

APPENDIX B
BDM AND BDM-RT API

A list of relevant functions belonging to the BDM and BDM-RT Application Programmers Interface (API) are listed below for easy reference.

B.1 BDM API Functions

BDM_Frame_malloc Allocates a send-buffer from a buffer queue.

BDM_Frame_send Sends a message.

BDM_Frame_recv Returns the first received message.

BDM_Frame_free Frees the receive-buffer into a buffer queue.

B.2 BDM-RT API Functions

BDMRT_Frame_malloc Allocates a send-buffer from a buffer queue.

BDMRT_Frame_send Sends a message.

BDMRT_Frame_recv Returns the first received message with a matching tag.

BDMRT_Frame_free Frees the receive-buffer into a buffer queue.

A Real-Time Message Layer over Myrinet Networks

By

Xinyan Zan

**A Project Report
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science**

Mississippi State, Mississippi

August 2000

A Real-Time Message Layer over Myrinet Networks

By

Xinyan Zan

Approved:

Dr. Anthony Skjellum
Associate Professor of Computer
Science
(Major Professor)

Dr. R. Rainey Little
Associate Professor of Computer
Science
(Committee Member and
Graduate Coordinator)

Dr. Donna S. Reese
Associate Professor of Computer
Science
(Committee Member)

Name: Xinyan Zan

Date of Degree: August 5, 2000

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: A Real-Time Message Layer over Myrinet Networks

Pages in Study: 35

Candidate for Degree of Master of Science

This report describes a real-time message layer over Myrinet networks. It is intended to provide communication support with QoS to upper level layers like MPI/RT. In order to minimize development efforts and utilize an existing communication layer's features, GM has been chosen as the basis of this project. GM is a Myrinet communication layer provided by the vendor of Myrinet gigabits networks – Myricom Inc. It is a powerful communication layer with a lot of features like automatic network mapping. In order to provide real-time communication, a resource reservation scheme is employed for real-time purpose. Corresponding changes are made to the memory management and link scheduling of GM for this purpose. The resulting system is composed of a modified GM system and a real-time memory management layer.

Tests are also conducted to verify the correctness of the system and the real-time performance. Conclusions are reached from the test results that the real-time traffic and non-real-time traffic are integrated successfully in the system.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major advisor Dr. Anthony Skjellum for his valuable guidance and support through my graduate program. I particularly appreciate his patience, kindness, and valuable suggestions to my project.

I would also like to thank Dr. R. Rainey Little and Dr. Donna S. Reese for serving on my committee and for providing valuable suggestions and comments.

Thanks also go to HPC group and help@myri.com for their cooperation and support.

My final but in no means the least thanks go to my parents for their unselfish support during my study. They have always been my strength source and inspiration although I am far away from home.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	
LIST OF TABLES	
LIST OF FIGURES	
CHAPTER	
I INTRODUCTION	
1.1 Background	
1.2 Motivation	
1.3 Organization	
II PROBLEM ANALYSIS AND CURRENT SOLUTION	
2.1 Problem analysis	
2.2 Resource reservation scheme	
2.3 RSVP	
2.3.1 RSVP introduction	
2.3.2 RSVP components	
2.3.3 RSVP current status	
2.4 RT-MACH at Carnegie Melon University	
2.4.1. introduction to RT_Mach	
2.4.2 reservation abstraction in RT_MACH	
2.5 Conclusion	
III SYSTEM ARCHITECTURE	
3.1 Overview of the system	
3.2 Software architecture	
3.2.1 RT_Linux	
3.2.2 Turtle scheduler	
3.2.3 Communication layer	
3.3 Hardware architecture	
3.3.1 Myrinet	

Chapter

IV GM ANALYSIS

4.1 Introduction to GM.....

4.2 GM analysis.....

4.2.1 GM memory management.....

4.2.2 Link scheduling in GM

4.2.3 Flow control and error control in GM.....

V IMPLEMENTATION.....

5.1 Memory management of real-time traffic

5.1.1 Host side memory.....

5.1.2 The Lanai side memory.....

5.1.3 Implementation detail.....

5.2 Link scheduling of real-time traffic.....

VI TEST RESULTS AND RESULTS ANALYSIS.....

6.1 Predictability test.....

6.1.1 Test metric.....

6.1.2 Test method

6.1.3 Tests to be conducted

6.1.4 Expected results.....

6.2 Test results.....

6.2.1 Test 1

6.2.2 Test 2

6.2.3 Test 3

6.3 Results Analysis

VII CONCLUSION AND FUTURE WORK.....

REFERENCES

APPENDIX

A. PROJECT CONTRACT

B. TEST RESULTS.....

Appendix

C. SOURCE CODE

LIST OF TABLES

Table

6.1 DETAIL DESCRIPTION OF TIMESTAMPS.....

6.2 AVERAGE ROUND-TRIP LATENCY AND JITTER OF TESTS

LIST OF FIGURES

Figure

<u>2.1 System Layout of a Resource Reservation System</u>	
<u>2.2 System Layout of RSVP</u>	
<u>3.1 System Architecture</u>	
<u>3.2 RT Linux Structure</u>	
<u>3.3 LANai 7 Network Interface Card</u>	
<u>4.1 Data Flow in GM</u>	
<u>5.1 Diagram of New System</u>	
<u>5.2 Dispatch Tables in GM</u>	
<u>6.1 Test Diagram</u>	
<u>6.2 Round Trip Latency of Single non-RT Traffic</u>	
<u>6.3 Round Trip Latency of Multiple non-RT Traffic</u>	
<u>6.4 Round Trip Latency of RT Traffic with no non-RT Traffic</u>	
<u>6.5 Round Trip Latency of RT Traffic with Single non-RT Traffic</u>	
<u>6.6 Round Trip Latency of RT Traffic with six non-RT Traffic</u>	

CHAPTER I

INTRODUCTION

1.1 Background

The QoS (Quality of Service) requirements of applications require that middleware provide guaranteed service. MPI/RT (Message Passing Interface/Real-Time) is a message-passing interface that is intended to provide guarantees for QoS for data communication functions (Cui et al. 1997). It is a real-time analogue of the well-known message passing standard - MPI. However middleware like MPI/RT cannot provide guaranteed service by itself. It must rely on support from the underlying operating system and communication layers. Some research has been done in this area to address this problem. Examples include FM-QoS at UIUC and BDM-RT in Mississippi State University. The objective of this project is to develop a real-time communication layer over Myrinet (a gigabits network). The resulting message layer should be able to provide guaranteed service to upper-level software such as MPI/RT.

1.2 Motivation

Unpredictability in a data transmission is caused by contention over shared resources. Contention will cause unpredictable waiting time. There are different ways to

solve the contention over shared resources. For example, FM-QoS uses a non-conflicting global schedule to avoid contention, to provide predictable service to the upper layer (Song and Chien 1999). But basically, all these ways use resource reservation schemes as a basic rule. A resource reservation scheme is composed of two parts: an admission control layer and a scheduler (Rajkumar et al. 1998). A detailed introduction to the resource reservation schemes will be given in next chapter. Since we already have a time-based scheduler working under RT-Linux, the objective of this project is to provide a low level communication layer to cooperate with the existing time-based scheduler in order to provide guaranteed service.

1.3 Organization

The organization of this report is as follows: Chapter 2 gives a literature review and some background knowledge. Chapter 3 gives the system architecture of this project. Chapter 4 gives a detailed analysis of GM. Implementation details are presented in Chapter 5. Finally, in Chapter 6, test results and results analysis are presented. Chapter 7 presents conclusions and future work.

CHAPTER II

PROBLEM ANALYSIS AND CURRENT SOLUTION

In the chapter, the problem needing to be solved in order to provide real-time communication will be analyzed. The current solutions to solve this problem will also be given.

2.1 Problem analysis

The fundamental issue in delivering quality-of-service (QoS) in network communication is resource management (Chien and Kim 1997). As mentioned in chapter 1, unpredictability in a data transmission is caused by contention over shared resources. Resource sharing will result in unpredictable waiting time. In order to provide predictable service to applications, the underlying operating system and communication sub-layer must employ a certain kind of mechanism to regulate the usage of such shared resources. Shared resources involved in a data transmission include CPU (protocol processing, packetization, etc), host memory, and network link bandwidth (Rajkumar et al. 1998). The task of any real-time communication layer is to solve the contention over these shared resources and make the whole transmission predictable. A lot of research has been done to solve this problem. Essentially, all these research efforts employ resource reservation schemes to address the contention.

2.2 Resource reservation scheme

A resource reservation scheme is designed to enforce the rule that before using the shared resources, a user must make a reservation for them first. A reservation is an indicator of what time and durations resource is needed (Rajkumar et al. 1998). According to the reservation request, the system will make the decision whether to accept this request (if the system can meet the requested QoS requirements) or decline this request (if the requested QoS requirements cannot be satisfied). Once the request is granted, the user is guaranteed to get the required services. Normally, there are two key components in resource reservation schemes: one is admission control (this is where the decision is made), another is the scheduler (this is the part to enforce the contract made by the admission control layer to user applications). The following figure shows the system layout in a resource reservation system.

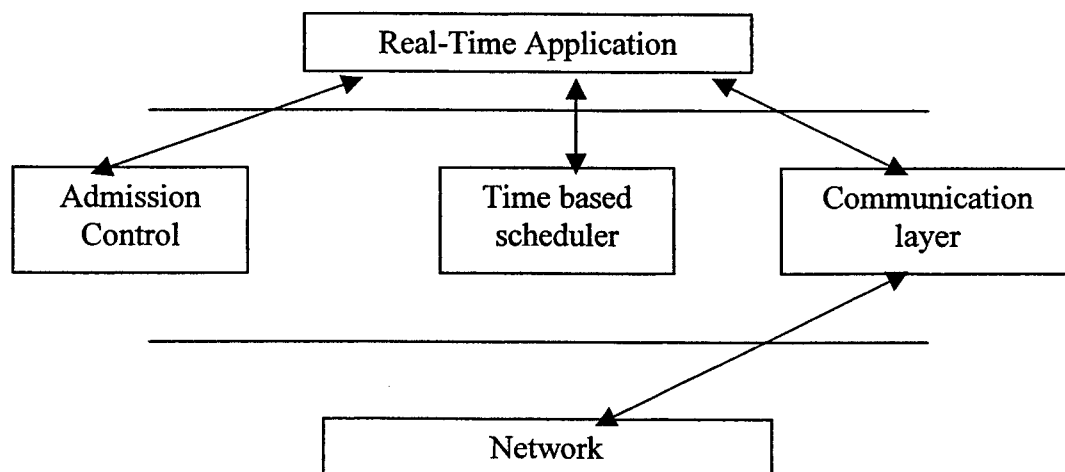


Figure 2.1 System layout of a resource reservation system

The resource reservation scheme has been employed in many circumstances that need guarantees on QoS. In this section, two examples are given. One is RSVP

(ReSource reserVation Protocol). Another is RT-MACH from Carnegie Mellon University.

2.3 RSVP

2.3.1 RSVP introduction

The resource reservation protocol is an internet protocol intended to provide QoS guarantees to end users (Stallings 1997). Network protocols like TCP/IP, UDP/IP, do not address any QoS requirements. When it comes to the applications that are QoS sensitive, like audio/video transmission, users will often get unacceptable transmission quality. RSVP (ReSource reserVation Protocol) is proposed to improve delivered quality of service under such circumstances. "It is a network control protocol that will allow Internet applications to obtain special qualities-of-service (QoS's) for their data flows" (Cisco Systems Inc. 1999). The essence of RSVP is to employ a resource reservation scheme in order to provide guaranteed service to user applications. RSVP is a transport layer protocol in the OSI seven-layer model. It can be used with both IPv4 and IPv6 (Cisco Systems Inc. 1999). The startup procedure of an RSVP session includes two phases (Stallings, 1997). First, the sender will send an RSVP path message to the destination. All the intermediate routers will keep a record of the route (including the previous router and next router). When the receiver receives this path message, it will send out a resource reservation request all the way back to sender. All the intermediate routers will allocate resources (include link bandwidth, buffers, CPU time) according to the flow specification in this request if the request can be satisfied. Otherwise, the RSVP program returns an error indication to the application that originated the request.

2.3.2 RSVP components

RSVP is composed of the following functional components. Each of these has a certain task. The first one is called the RSVP daemon which is the main module of RSVP. It is responsible for asking the "policy control" and "admission control" for their permission to set up the reservation. If permission is granted, the RSVP daemon sets parameters in a "packet classifier" and "packet scheduler" in order to obtain the desired QoS (Cisco Systems Inc. 1999).

Policy control is used for authority checking. It will check if the requester is allowed to make reservations and what kind of QoS he or she may reserve.

Admission control is used to evaluate the QoS requirement in the reservation request. It will check the current state of the node (host or router) to see if the node has sufficient resources to meet the requested QoS requirement.

The packet classifier is responsible for determining the route and QoS class for each incoming data packet.

The packet scheduler is responsible for scheduling queued packets according to their QoS parameters. This is the part responsible for achieving the promised QoS. This can be implemented by prioritizing queues of flows, as necessary.

The system layout of RSVP can be described by following figure.

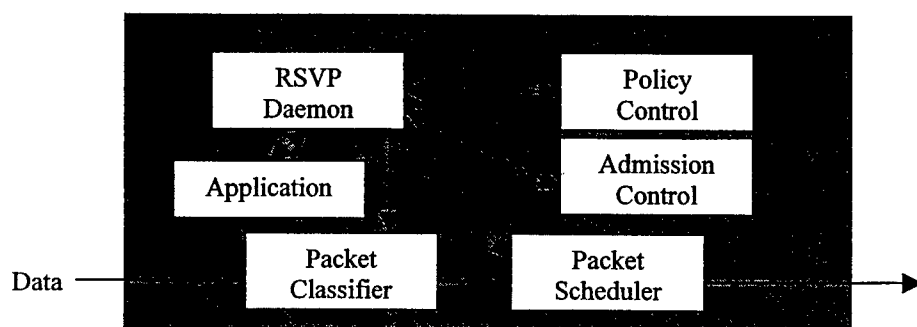


Figure 2.2 System layout of RSVP (Cisco Systems Inc. 1999)

2.3.3 RSVP current status

RSVP was originally conceived by researchers at the University of Southern California Information Sciences Institute and Xerox Palo Alto Research Center (Cisco Systems Inc. 1999). The Internet Engineering Task Force (IETF) is now working toward standardization through an RSVP working group (Cisco System Inc. 1999). At present, many vendors of operating systems and routers are incorporating RSVP and integrated services into their products for near-future availability. The potential for the use of RSVP is enormous, and it is growing as more and more interactive multimedia applications are presented for use over the Internet or organizational intranet (Stallings 1997).

2.4 RT-MACH at Carnegie Melon University

2.4.1. Introduction to RT_Mach

RT_MACH is a real-time operating system developed at Carnegie Mellon University. It is an extension to the Mach operating system (Rajkumar et al. 1998). It was developed under the circumstance that the general time-sharing operating system cannot satisfy multimedia applications' (such as video and audio) timing requirements. It is a micro kernel based operating system which means only the most basic functions are put into the kernel, all the other functions are put outside the kernel as servers. RT_MACH supports multiple scheduling policies, such as rate monotonic scheduling and earliest deadline first scheduling. The resource reservation abstraction in RT_MACH is an implementation of a resource reservation scheme.

2.4.2 Reservation abstraction in RT_MACH

In reservation abstraction, the system considers each shared resources (physical or logical) such as CPU, memory and semaphore as a kind of resource (Rajkumar et al.

1998). When a real-time task requests a resource and the request is granted, a reservation of this resource is saved for this task. The system guarantees the availability of the resource when the task wants to use it. This kind of scheduling is called resource-centric scheduling (Chen, Rajkumar, and Mercer 1996). RT_MACH describes each task by five parameters: start time, end time, period, deadline and computation time (Chen, Rajkumar, and Mercer 1996). The timing requirement of each task is to use requested resource for computation time before deadline in each period. Each guaranteed task will get its reservation and not influence other tasks.

2.5 Conclusion

In this chapter, we analyzed the problem that needs to be solved in order to provide guaranteed service. We also analyzed a solution to this problem – the resource reservation scheme. Two implementations of resource reservation scheme (RSVP and RT_MACH) have been given.

CHAPTER III

SYSTEM ARCHITECTURE

In this chapter, the system architecture used in this project is given. We first give an overview of the system. Then a detailed analysis of each component will be given.

3.1 Overview of the system

Figure 3.1 shows an overview of system architecture.

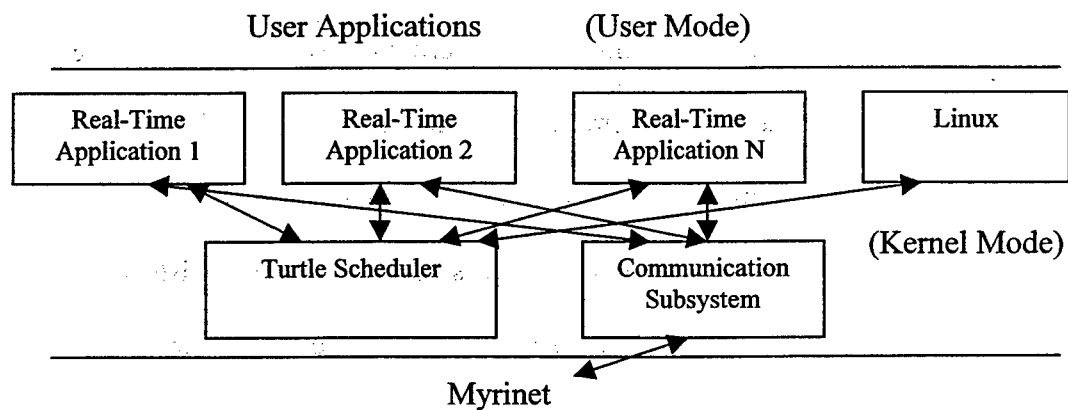


Figure 3.1 System Architecture

3.2 Software Architecture

3.2.1 RT_Linux

RT_Linux is chosen as the operating system. Linux is an increasingly popular free Unix-clone operating system (Atlas et al. 1998). Because it is free and all the source code is available, some research has been done on Linux to make it into a real-

time operating system that can support hard real-time application. RT_Linux is developed under such background. It was implemented at New Mexico Tech. The basic idea is to change Linux into a real-time OS without changing the kernel too much (Barabanov and Yodaiken 1996). Standard Linux cannot support hard real-time applications because:

- The Linux kernel is non-preemptive. When the user makes a system call and it goes into the kernel mode, Linux will lock the scheduling until the system call finished. At this time, a ready task with higher priority cannot preempt the lower priority task that is running the system call. This may cause unpredictable latency for a real time task.
- In the critical section, Linux kernel always uses *cli* to disable interrupts to keep the consistency of critical resources. Like scheduling lock, this is also a problem for real-time tasks. What RT_Linux does is to add a real-time kernel over Linux and make Linux as a real-time task with the lowest priority. Linux is not aware of the existence of a RT kernel. The following figure shows the structure of RT_Linux.

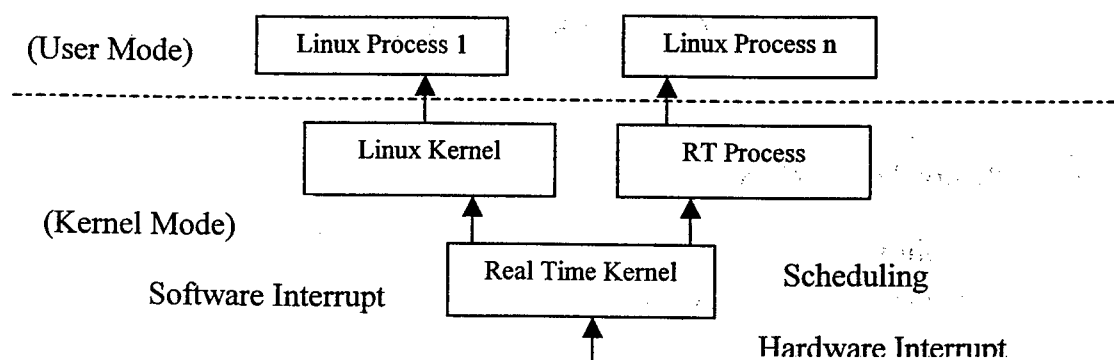


Figure 3.2 RT_Linux structure

From the Figure 3.2, we can see that all real time tasks and real time kernel are running in the kernel mode. Only the Linux processes are running in user mode. Another big change of RT_Linux comparing to Linux is that all the *sti* and *cli* instructions in the kernel are replaced by soft *sti* and *cli*. What the soft *sti* and *cli* do is not directly set or clear the status register in the CPU, instead they only set or clear the value of a soft flag. In the interrupt emulator, it will check this soft flag to see if it should pass current interrupt to Linux kernel or not. All the real-time interrupts are responded to immediately, thus avoiding unpredictable interrupt latency in Linux. There are also some drawbacks of RT_Linux. First, putting the real time task into the kernel is sometimes dangerous. There is no spatial protection between the kernel and the application. A logic error in the real time application may cause the whole system to crash. Second, the Linux kernel system primitives are not reentrant in nature and RT_Linux does not change this point. To keep system consistencies the real time task cannot use any system services provide by Linux kernel. Third, two scheduling policies are implemented in current RT_Linux. One is a fixed priority based preemptive scheduler, another is earliest deadline first (EDF) scheduler. Both of these do not provide temporal protection among real time tasks. A time-misbehaving real-time task can easily influence the run time behavior of other normal real time tasks. The last one is not a problem for RT_Linux. Because the user can implement his or her own schedulers and add it to the kernel easily. The Turtle scheduler is such a scheduler implemented on RT_Linux (Apte et al. 1999).

3.2.2 Turtle scheduler

Turtle scheduler is a real-time scheduler on RT_Linux developed in the HPC group of Mississippi State University. It adopted the reservation abstraction in

scheduling policy. The scheduling algorithm it uses is critical deadline (deadline-computation) first algorithm (Apte et al. 1999). It is similar to Earliest Deadline First algorithm. Each periodic hard real time task is represented by five parameters: C_i , P_i , D_i , S_i , E_i . They represent computation time, period, deadline, start time and end time of a periodic hard real time task respectively. The scheduler always chooses the task with the earliest critical start time to run. The deadline driven scheduling algorithm is optimum in the sense that if a set of tasks can be scheduled by any algorithm, it can be scheduled by the deadline driven scheduling algorithm (Liu and Layland 1973). The scheduler only guarantees the requested time of a real time task. The scheduler keeps track of the run time of a real time task. When a real time task overruns its requested time and another real time task is ready to run, the scheduler will get the overrun task out of the CPU and let the other ready real time tasks run. In the real-time field, this is called a reservation abstraction. Just as the virtual memory mechanism will provide spatial protection among processes, the reservation abstraction can provide temporal protection among real time applications, which means a runtime misbehaving task will never influence other normal real-time tasks runtime behavior.

3.2.3 Communication layer

The task of the communication layer is to cooperate with the real-time scheduler in order to provide guaranteed service to communication tasks. The objective of this project is to provide such a communication layer in order to achieve real-time communication. A detailed analysis of the communication layer will be given in the next chapter.

3.3 Hardware architecture

A two-node Myrinet cluster is used in the experiments. Both of hosts are Pentium Pro 200. Myrinet is used as network test bed. A detailed introduction to Myrinet is given below.

3.3.1 Myrinet

Myrinet is a high performance gigabit per second network. It can provide full duplex data transmission rate at 1.28 Gigabit/second (Boden et al. 1997). It uses cut-through switch to provide low latency data transmission. Myrinet is also really reliable. It exhibits a very low bit-error rate, less than one bit error per day in a large network, and is highly robust with respect to host, switch, and cable faults (Myricom Inc. 1999). A Myrinet network is composed of three parts: Myrinet network interface cards, cables and switches.

A Myrinet interface card is a programmable card. It has an embedded processor (LANai) on it. The control program running on this embedded processor is to handle direct interaction with host and network and is called the MCP (Myrinet Control Program). The user can implement his or her own communication layer over Myrinet by writing a customized Myrinet Control Program. There are three DMA engines on the LANai (Myricom Inc. 2000). The first one is responsible for the data transmission between host and LANai. The second one is responsible for data transmission from LANai to network. The third is responsible for data transmission from network to LANai. The diagram of a Myrinet network interface card is shown in Figure 3.3.

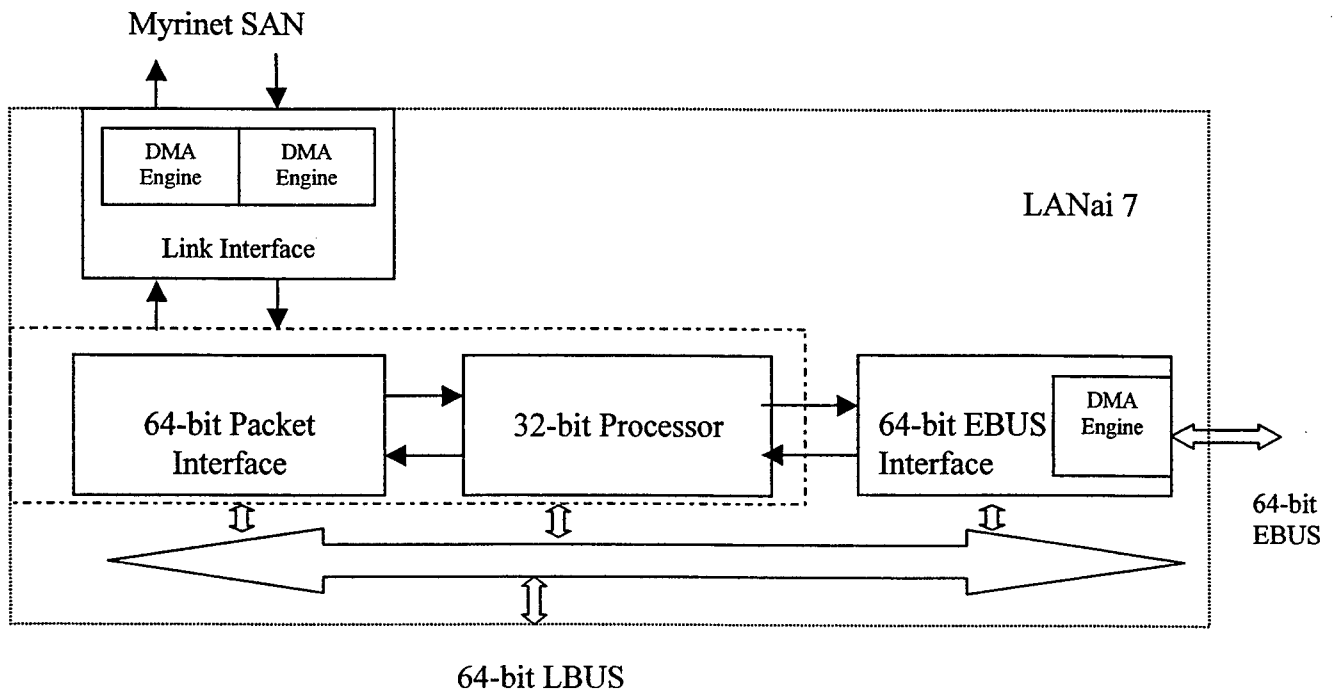


Figure 3.3 LANai 7 Network Interface Card (Myricom Inc. 2000)

CHAPTER IV

GM ANALYSIS

Instead of building a whole new communication layer, GM (Glenn's Messaging) was chosen as the basis for this project. GM is composed of an MCP, a host library and a device driver. In this chapter, a detailed analysis of GM will be given.

4.1 Introduction to GM

GM is an open source, high performance message layer over Myrinet. GM's design objectives included low computational overhead, portability, low latency, and high bandwidth (Myricom Inc. 2000). It is provided by the vendor of Myrinet, Myricom Inc. There are several features in GM that make us choose it as our basis.

- Open source (So we can modify)
- Low latency
- Automatically maps network.

The GM system includes a driver, Myrinet interface firmware, a network mapping program and the GM API, library, and header files. By choosing GM as the basis for this project, it is possible to utilize a lot of features provided by GM, like network mapping etc. In this way, we can get a real-time communication layer with minimum extra development effort.

4.2 GM analysis

The problem we are facing is that GM is a high performance message layer, not a real-time one. By analyzing the memory management and link scheduling in GM, we found out the memory management and link scheduling between real-time traffic and non-real-time traffic must be separated. The following graph shows the data flow in GM.

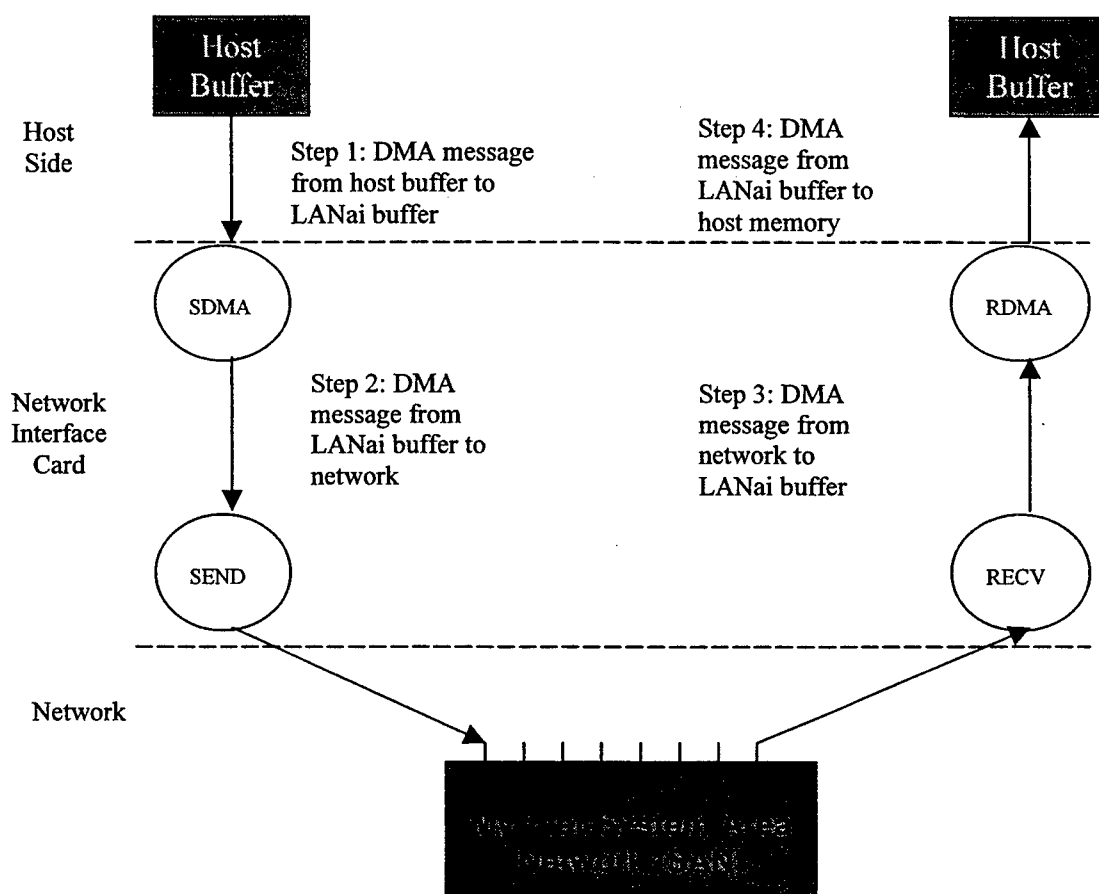


Figure 4.1 Data flow in GM

From this diagram, one can see that the data transmission in GM is finished via four DMA transfers. On the sender side, the sender needs to DMA data from host buffer to LANai buffer first. Then the MCP on sender side will DMA these data from LANai

buffer to network. On the receiver side, the MCP will DMA data from network to LANai buffer, then finally, DMA these data from the LANai buffer to host buffer.

4.2.1 GM memory management

As the graph shows, for data transfer between host and LANai, GM currently only supports DMA transfer. Before sending out or receiving any data, GM process must get the memory both in the host and on the LANai.

Host side memory management

Before sending and receiving data from the LANai, GM process must get a DMAable (contiguous and non-pageable) memory from kernel. `Gm_dma_malloc` is used for this purpose. What it does is to use `kmalloc()` for kernel port and use `malloc()` for user port to get DMAable memory. This is not appropriate for real-time use, since it will cause unpredictable waiting time when there is not enough memory available.

LANai buffer management

The MCP in GM only has two send buffers and two receiver buffers (Myricom Inc. 2000). When there is a send arriving from host or a receiver arriving from network, and there is no available buffer, the MCP will put the send and receive into a waiting queue, until the buffer becomes available. As with the memory management on host side, this will also cause unpredictable waiting time, and thus is not appropriate for real-time purposes.

4.2.2 Link scheduling in GM

The GM communication system provides reliable, ordered delivery between communication endpoints called "ports," with two levels of priority (Myricom Inc. 2000). Before sending or receiving data from the Myrinet network, the user must open a port

using `gm_open()`. Port number, destination node number and priority uniquely identify a sub-port. The send queue in GM is a circular list of connections with pending sends. Each connection maintains a list of sub-ports that have sends pending over that connection. Each sub-port maintains a list of send tokens describing packets that have data to be sent, or have acknowledgements pending. For fairness, in GM after each send, the control program rotates the sub-port queue for the connection, and then rotates the connection queue (Myricom Inc. 2000). This is a kind of time-sharing scheduling policy. It is not QoS sensitive, thus not appropriate for real-time purpose.

4.2.3 Flow control and error control in GM

GM uses the "go back N" protocol with NACKs in flow control and error control (Myricom Inc. 2000). "Go back N" is the preferred protocol when software overhead, rather than network capacity, is the network's limiting factor, because it wastes network bandwidth during error recovery to reduce software overhead relative to other protocols (Tanenbaum 1996). This is unnecessary for real-time traffic, since retransmission in real-time traffic is most often not desired. This is also the reason why hard real-time traffic always requires high reliability from underlying network.

CHAPTER V

IMPLEMENTATION

After analyzing GM, the conclusion was reached that the memory management and link scheduling of real-time traffic and non real-time traffic must be separated. In this chapter, implementation details of real-time traffic memory management and link scheduling are given.

5.1 Memory management of real-time traffic

5.1.1 Host side memory

The solution is to build a RT module over the GM device driver. When this RT module is loaded, it will pre-allocate a certain amount of memory from the kernel. Real-time tasks will get and release memory by making requests to this RT module. After obtaining memory from kernel, this RT module will organize this piece of memory into a circular buffer.

5.1.2 The Lanai side memory

We use shadow buffers on LANai. Once a communication task gets a buffer in host or in LANai, it also gets the corresponding buffer in LANai or host. DMA transfers data between corresponding buffers in LANai and host.

5.1.3 Implementation detail

On the host side, there are two options to pre-allocate buffer from kernel. One is to use `kmalloc (size, GFP_DMA)` to get non-pageable, contiguous memory from kernel. Another is to use memory map function provided by Linux kernel to map high address memory into kernel address space. In this case, we choose the later option. This is based on the concern that: memory obtained by using `kmalloc (size, GFP_DMA)` belongs to the low 16MB memory. Memory in this range is very precious. If too much DMAable memory is allocated, other `kmallocs` in the system may fail. By using `vremap()`, we can overcome this drawback.

On the LANai side, send and receive buffers are allocated by static array. A static array of size `RT_MAXFRAME*(RT_MAXSEND+RT_MAXRECV)` is added into the GM data structure.

5.2 Link scheduling of real-time traffic

For link scheduling between real-time traffic and non real-time traffic, priority-based scheduling is used. The new system diagram is shown in Figure 5.1. Real-time events always have higher priority over non-real-time events. In this way, we can guarantee that real-time traffic will always be scheduled ahead of non-real-time traffic. For link scheduling between real-time traffic, First-In-First-Out (FIFO) scheduling is used. This is based on the assumption that the arriving order of real-time packets is already scheduled by the scheduler on the host side.

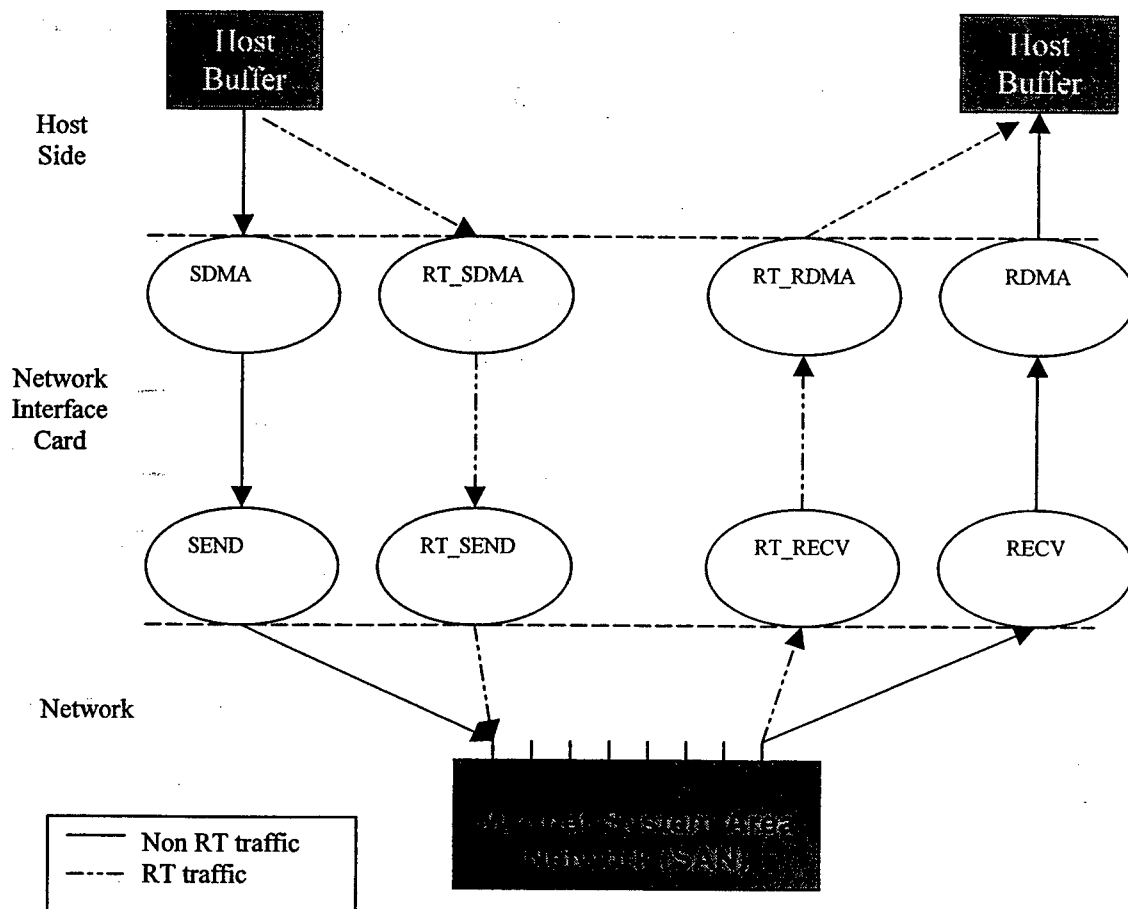


Figure 5.1 Diagram of new system

In GM, the system status is expressed by ISR&IMR|GM_STATE (Myricom Inc. 2000). The hardware information, such as whether the send interface is ready or not, is expressed by the hardware register ISR. It is maintained by the LANai hardware. IMR is the interrupt mask register. Other information, such as whether there is a packet pending to be sent, is expressed by a 32 bit global variable GM_STATE that is set and cleared by software (Myricom Inc. 2000). The dispatch of GM can be finished within two instructions. Two tables are set to achieve this goal. The first one is called the event index table. It is responsible for converting current status (ISR&IMR|GM_STATE) to

the corresponding event index. The second table is called the handler table. It is responsible for converting the event index to the corresponding handler address. The dispatch procedure is expressed by Figure 5.2.

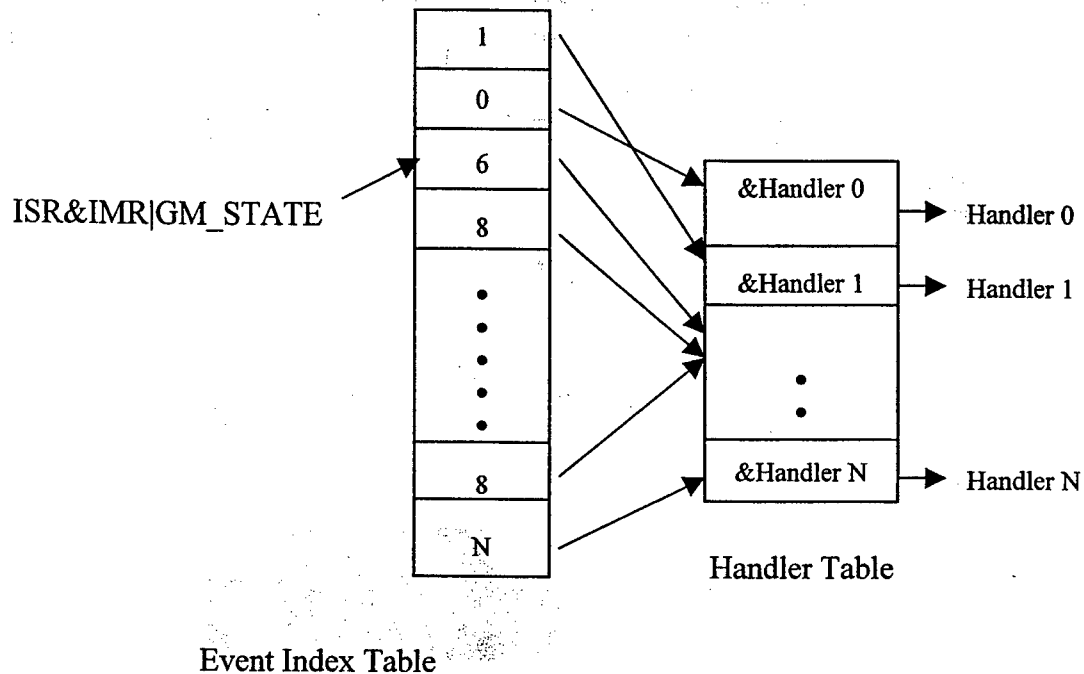


Figure 5.2 Dispatch tables in GM

In order to add real-time ability to GM, another 8 events are added into system. They are as follows: `START_RT_SEND_EVENT`, `FINISH_RT_SEND_EVENT`, `START_RT_SDMA_EVENT`, `FINISH_RT_SDMA_EVENT`, `START_RT_RECV_EVENT`, `FINISH_RT_RECV_EVENT`, `START_RT_RDMA_EVENT`, `FINISH_RT_RDMA_EVENT`. Correspondingly, another 8 status bits are added into system. They are as follows: `RT_SDMAING`, `RT_SDMA_PENDING`, `RT_SENDING`, `RT_SEND_PENDING`, `RT_RDMAING`, `RT_RDMA_PENDING`, `RT_RECEIVING`, `RT_RECV_PENDING`. In the original GM system, only 15 bits are used to express

currently system status. The size of event index table is 32K byte. Each time we add a status bit, the event index table will double in size. Since 8 status bits are added, the size of event index table will be 8 MB. This is not realistic for an embedded processor since the memory of an embedded system is always precious. In our case, the maximum SRAM for a LANai processor is 4MB. Because of the limitation of memory, for real-time events, we use *if* statement instead of table dispatching. Performance is sacrificed for memory usage.

CHAPTER VI

TEST RESULTS AND RESULTS ANALYSIS

6.1 Predictability Tests

6.1.1 Test Metric

In order to test the predictability of the system, latency variance (jitter) is used as the test metric. It is an important metric to measure the performance of a real-time system.

6.1.2 Test method

A 32-bit timer on LANai board is used to get timestamps. The resolution of this RTC (Real-Time Clock) is $0.5 \mu s$.

Figure 6.1 shows the complete test procedure.

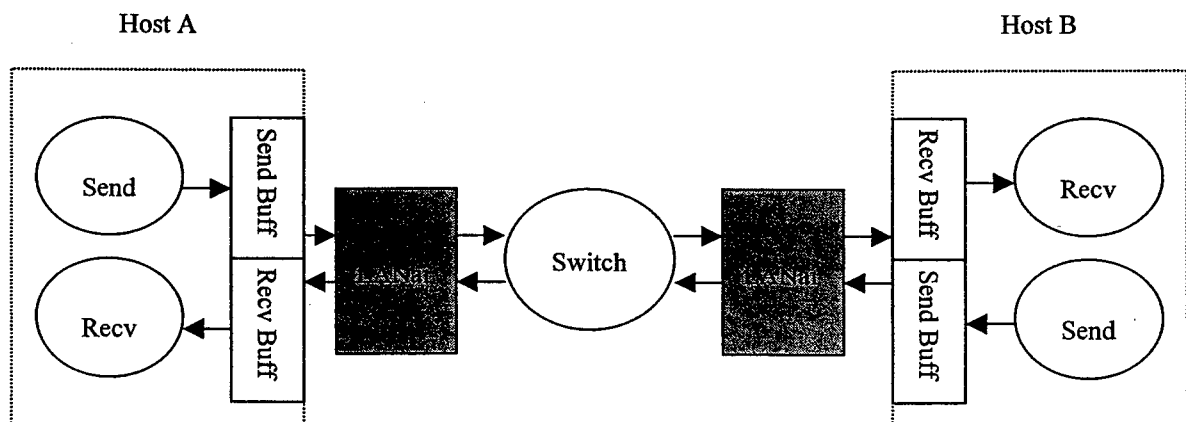


Figure 6.1 Test Diagram

Round trip latency is tested because of two concerns. The first one is that currently there is no global clock synchronization support in the MCP. The second one is that if one uses global clock synchronization, the accuracy of test results will heavily depend on the accuracy of global clock synchronization.

The test plan is as follows. There are four real-time tasks running on two nodes. The send task on node A will send a message to host B. When the receive task on host B receives this message, it will pass the message to a send task on host B. Then the send task on host B will send this message all the way back to host A. During this procedure, one gets eight timestamps. By computing these timestamps using the following formula, we can get the round trip latency.

$$\text{Round-Trip Latency} = (\text{time8} - \text{time1}) - (\text{time3} - \text{time2}) - (\text{time5} - \text{time4}) - (\text{time7} - \text{time6})$$

Table 6.1 Detail description of timestamps

Time	Description
Time 1	The time when send task on node A send out the message
Time 2	The time when LANai on node B gets the message
Time 3	The time when receive task on node B gets the message
Time 4	The time when receive task on node B send this message to send task on node B
Time 5	The time when send task on node B starts to send back the message
Time 6	The time when LANai on node A gets the sending back message
Time 7	The time when receive task on node A gets the sending back message
Time 8	The time when receive task on node A finish computation

6.1.3 Tests to be conducted

Test 1: There is no real-time traffic in the system. Latency jitter of the "original" GM traffic is tested. By increasing the number of communication tasks, the impact of shared resource contention on latency variance can be observed.

Test 2: There is only real-time traffic in the system. Latency jitter of real-time traffic is tested.

Test 3: Both real-time traffic and non real-time traffic is in the system. Latency jitter of real-time traffic is tested to see the impact of non real-time traffic on real-time traffic.

6.1.4 Expected results

The results of test 1 (jitter of non-real-time traffic) should be high, since there is no mechanism used to guarantee the predictability of message transfer.

The results of test 2 (jitter of RT traffic) should be small. It should be much better compared to the jitter of non-real-time traffic.

The results of test 3 (jitter of RT traffic with the presence of non-real-time traffic) should be close to the results of test 2, since ideally real-time traffic should not be influenced by non-real-time traffic.

6.2 Test results

6.2.1 Test 1

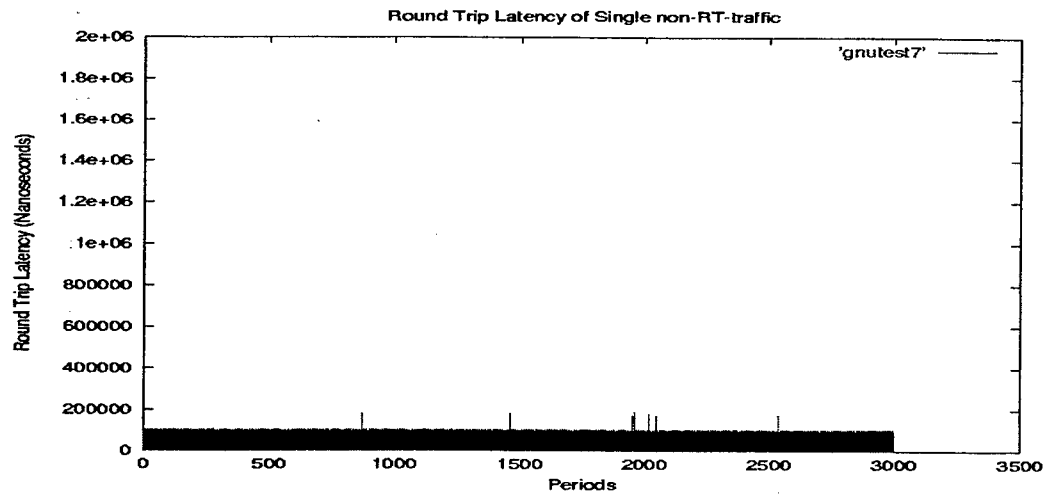


Figure 6.2 Round Trip Latency of Single non-RT traffic

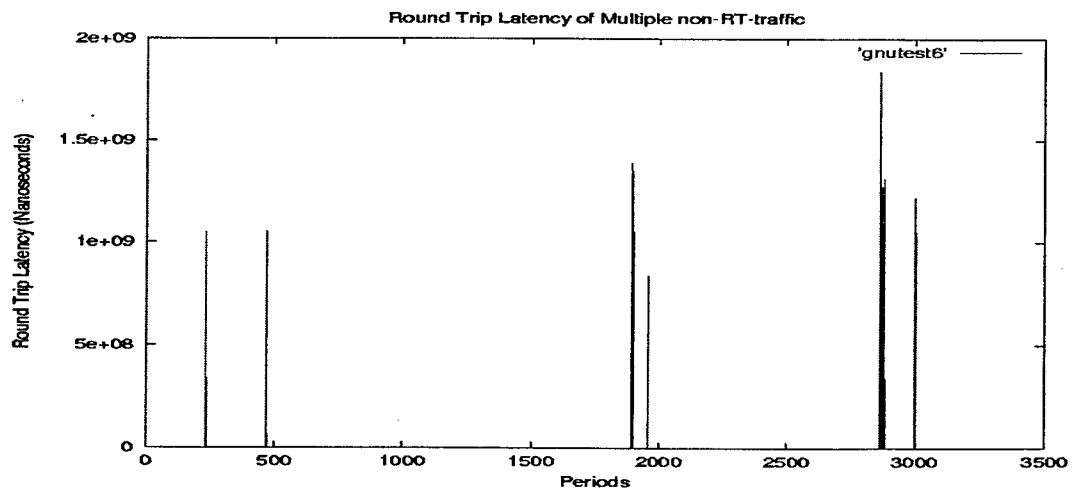


Figure 6.3 Round Trip Latency of Multiple non-RT traffic

6.2.2 Test 2

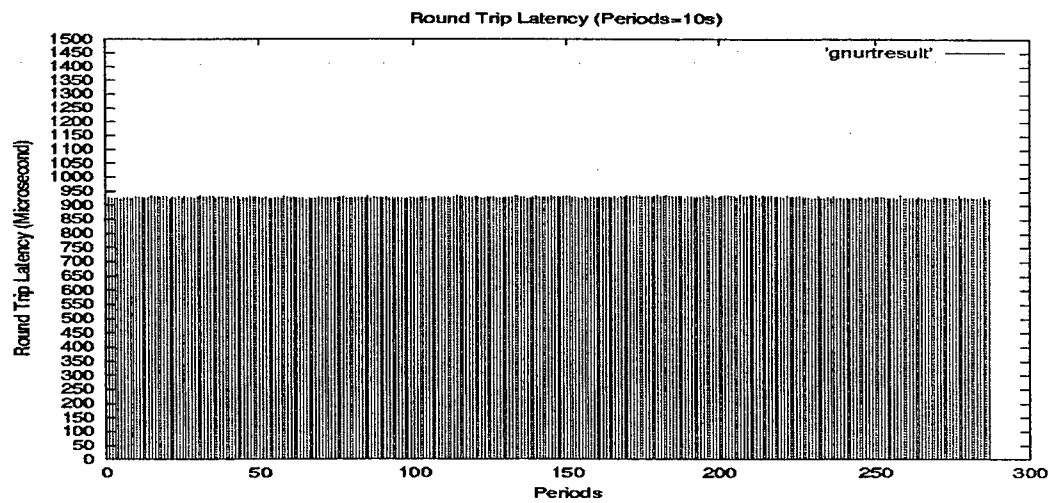


Figure 6.4 Round Trip Latency of RT traffic with no non-RT traffic

6.2.3 Test 3

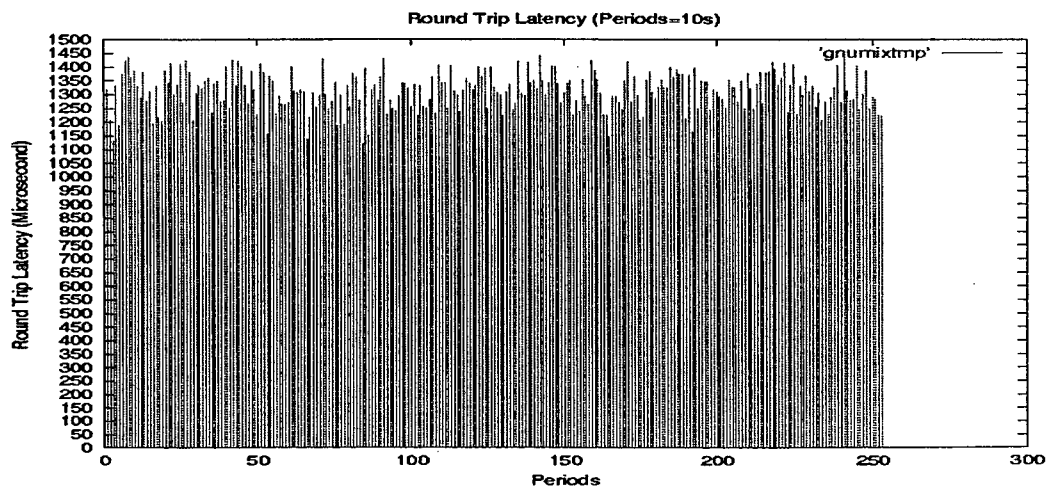


Figure 6.5 Round Trip Latency of RT traffic with single non-RT traffic

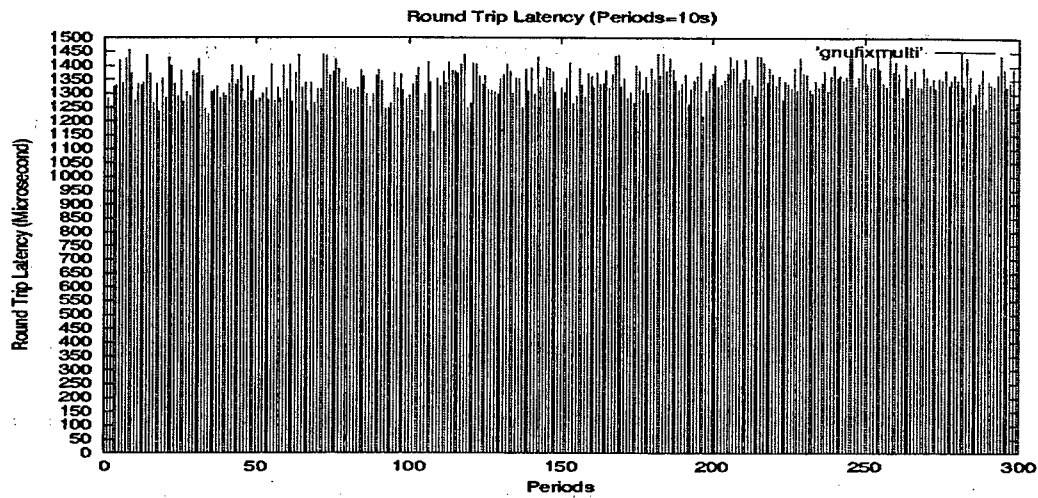


Figure 6.6 Latency jitter of real-time traffic with six non-RT traffic

Table 6.2 Average round-trip latency and jitter of tests

	Average Round-Trip Latency (Microsecond)	Jitter (Microsecond)
Single non-RT traffic	974.65	108.53
Six non-RT traffic	16488.472865423	2147410.962
RT traffic	926	17
RT traffic with single non-RT traffic	1312	322
RT traffic with six non- RT traffic	1345	296

6.3 Results Analysis

Figure 6.2 shows the round trip latency for single non-real-time traffic. From this figure, one can see that there is a long round trip latency appearing irregularly. The

biggest jitter observed so far is around 90 microseconds. This also matches the expected result. Since there is no mechanism used in the original GM to provide guaranteed services, latency jitter is expected to be high. Figure 6.3 shows the round trip latency for six non-real-time traffic. From this figure, we can see the impact of contention over shared resources on round trip latency. In this experiment, six GM communication peers are set up on two nodes. Figure 6.3 is the round trip latency of one peer. The biggest jitter in this case is more than 2 seconds. By comparing Figure 6.3 with Figure 6.2, we can draw the conclusion that the round trip latency of non-real-time traffic will increase when the workload in network increases. Figure 6.4 shows the round trip latency of single real-time traffic. The biggest jitter observed so far is only around 20 microseconds. This also matches the expected results very well. The mechanisms used in the system for real-time traffic do have an effect. Figure 6.5 shows the round trip latency of single real-time traffic with the presence of one non-real-time communication peer. From this figure, we can see that both the round trip latency and latency jitter increase in the presence of non-real-time peers. There are two reasons for this increase. The first one is that transactions on each state machine are non-preemptive. Dispatch occurs only after transactions. The second is that DMA operations on LANai are non-preemptive. One cannot initiate a DMA operation until the current DMA operation finishes. These two reasons explain why both round trip latency and latency jitter increase when there is non-real-time traffic in the system. Figure 6.6 shows the round trip latency of real-time traffic with the existence of six non-real-time communication peers. By comparing Figure 6.5 and 6.6, one can see that both round trip latency and latency jitter stay stable. They will not increase when the number of non-real-time

communication peer increases. Average round-trip latency increases. Since the possibility that a RT traffic waits for an on-going non-RT traffic increases, average round-trip latency will increase. Since the increase of worst case round-trip latency is due to the granularity of preemption, the worst case round-trip latency when there are multiple non-RT traffic should be the same as the worst case round-trip latency when there is only one non-RT traffic in the system. This explains why jitter stays stable in Figure 6.5 and Figure 6.6. Actual results match expected results. Table 6.2 shows the average round trip latency and jitter of these tests.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

The objective of this project is providing a real-time communication layer to cooperate with the existing scheduler to achieve real-time communication. In order to minimize the development efforts and utilize the existing communication layer's features, GM was chosen as the basis of this project. The problem needing to be solved in this project was how to provide real-time communication while still keeping the functionality of original GM. Current research in this area is studied in chapter two. The goal is achieved by separating memory management and link scheduling of real-time traffic and non-real-time traffic in this project. System architecture, system analysis and implementation details are given in chapter three, four and five respectively. Tests were also conducted to verify the functional correctness and real-time performance of the system. The complete results of tests conducted are presented and analyzed in Chapter six. The actual results match well with the expected results.

From the results of tests we have conducted, the following conclusions can be reached:

- We successfully integrated real-time traffic and non real-time traffic. The system functions correctly. Both real-time communication and non-real-time communication can run in the system simultaneously.

- System can cooperate with the existing scheduler (Turtle) to provide guaranteed service for real-time applications.

Although the goal of this project was achieved, the resulting system is not a complete system yet. Following is the work need to be done in the future.

- The current MCP does not support global clock synchronization. In order to support time-based scheduling, global clock synchronization support should be added in the future system.
- The final system should include admission control. This is a very important part in resource reservation scheme.

REFERENCES

- Apte, M., S. Chakravarthi, A. Skjellum, and X. Zan. 1999. Time-based linux for real-time Nows and MPI/RT. Submitted to the real-time systems symposium.
- Atlas, A. K., and A. Bestavros. 1998. Design and implementation of statistical rate monotonic scheduling in KURT Linux. <http://www.cs.bu.edu/techreports/98-013-sms-linux-implementation.ps.Z> (Accessed 20 November, 1999)
- Barabanov, M., and V. Yodaiken. 1996. Real-Time Linux. <http://www.rtlinux.org/rtlinux.new/documents/papers/li.ps> (Accessed 26 November, 1999)
- Boden, N. J., D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. 1995. Myrinet: A gigabit-per-second local area network. *IEEE Micro* 15(1): 29-36.
- Cisco Systems Inc. 1999. Resource reservation protocol (RSVP) http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/rsvp.htm (Accessed 16 January, 2000)
- Cui, Z., A. Kanevsky, J. Li, and A. Skjellum. 1997. MPI/RT: Design and implementation of a real-time message passing interface. http://www.mpirt.org/documents/pdpta97_revised.ps (Accessed 06 October, 1999)
- Lee, C., R. Rajkumar and C. Mercer. 1996. Experiences with processor reservation and dynamic QoS in real-time Mach. In *Proceedings of Multimedia Japan 96 held in Japan, April, 1996*.
- Myricom Inc. 1999. GM. <http://www.myri.com/GM/doc/gm.pdf> (Accessed February 26, 2000)
- Myricom Inc. 1999. LANai 7. <http://www.myri.com/vlsi/LANai7.pdf> (Accessed 06 January, 2000)
- Myricom Inc. 2000. PCI64 programmer's documentation. <http://www.myri.com/myrinet/PCI64/programming.html> (Accessed February, 2000)

- Pakin, S., M. Lauria, and A. Chien. 1996. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. <http://www.csag.ucsd.edu/papers/myrinet-fm-sc95.ps> (Accessed 26 November, 1999)
- Rajkumar, R., K. Juvva, A. Molano, and S. Oikawa. 1998. Resource kernels: A resource-centric approach to real-time and multimedia systems. <http://www.cs.cmu.edu/afs/cs/user/kjuvva/www/publications.html> (Accessed 26 January, 2000)
- Song, H. J., and A. Chien. 1999. Feedback-based synchronization for QoS traffic in cluster computing. <http://www.csag.ucsd.edu/papers/song.ps> (Accessed 20 December, 1999)
- Stallings W. 1997. *High speed networks: TCP/IP and ATM design principles*. Englewood cliffs, NJ: Prentice Hall
- Tanenbaum, A. S. 1996. *Computer networks*. Englewood cliffs, NJ: Prentice Hall

Final Report from the University of Maryland for
DARPA Contract 96144601

Ashok Agrawala
System Design and Development Group
Department of Computer Science
University of Maryland
College Park, MD 20742

September 10, 1999

This is the final report for the work done by the System Design and Development Group at the University of Maryland, College Park, as a subcontractor for Mississippi State University, for DARPA Contract 96144601.

1 Summary

The objective for our work in this contract was to enhance the Maruti operating system in several ways, in order to provide Mississippi State University with a platform upon which their work on the Real-Time Message Passing Interface could be developed. Maruti is a hard real-time operating system that has been in development at the University of Maryland for the past few years. For this particular contract, our goals were to enhance Maruti in two major and one minor areas. The two major areas are the high accuracy in the dispatching of real-time tasks and the dynamic time-based scheduling scheme. The minor area is in the development of a graphical tool to aid in the configuration and integration of hard real-time applications.

2 Real-Time Dispatching

In this area of research, our goals were to find ways to dispatch real-time tasks as close to their scheduled dispatch time as possible. This problem is difficult due to the relative unpredictability of various aspects in a computing system such as pipelining, cache hit/miss, etc. More specifically, we look at the accuracy of two approaches to dispatch real-time tasks. We start by studying the traditional approach of using count-down timers and discuss the factors affecting its precision. Next, we study a new approach for the deployment of the count-down timers that promises a higher degree of dispatching precision for real-time tasks.

The typical approach for the deployment of count-down timers is to preset the timer with a specific value representing the number of clock ticks required to elapse before a timer interrupt is generated. Till that time, the CPU can be used to run non real-time tasks. On the arrival of the timer interrupt, the CPU launches an Interrupt Service Routine (ISR) which handles the context switching from the current task to the real-time task.

In our implementation, we make use of the count-down timer present in the Advanced Programmable Interrupt Controller (APIC) which is part of the Intel® P6 Architecture. The P6 architecture is implemented in all Pentium Pro® and newer Intel microprocessors. In these systems, the system bus typically runs at a clock cycle of 66.667 MHz. The CPU multiplies the bus clock to synthesize its internal CPU clock.

The APIC timer has a 32-bit register that is decremented at each bus clock tick. For example, the counter is decremented every 15 nanoseconds when driven by a 66-MHz bus clock. The timer can be programmed to operate in one of two modes; "periodic", or "one-shot". In the periodic mode, the timer generates an indefinite number of interrupts periodically as determined by the initial value written in its register. Whereas in the one-shot mode, the timer generates only one interrupt.

Due to the limited width of the countdown register, the frequency of the bus clock, as well as other implementation-specific considerations, the longest period to program the timer may be limited to a maximum of M seconds. A typical value of M is 1.431 seconds for a 300 MHz Pentium II driven by a 66 MHz bus clock. If a longer period, say T , is needed, the timer may be programmed in periodic mode for $\lfloor \frac{T}{M} \rfloor$ iterations of length M seconds each, followed by a one-shot iteration of length $(T \text{ modulo } M)$ seconds.

Our analysis of the single interrupt approach has suggested that it is possible to reduce the amount of variability in the dispatching time through the absorption of the cache-miss effect. Pre-loading the CPU's level-1 cache with the required data and instructions has shown to significantly reduce the variability in the dispatching time. The technique that we use applies a double-interrupt approach to pre-load the data and instructions of that portion of the interrupt service routine responsible for the final dispatching procedures.

As with the single interrupt approach discussed above, the APIC timer may be programmed to generate few periodic interrupts, denoted by $APIC_n, \dots, APIC_2$. A semi-final interrupt, denoted by $APIC_1$, is generated at $t_r - \tau$ to pre-load the level-1 cache with the data and instructions required for the final dispatching steps. Fundamentally, $APIC_1$ touches all the memory paragraphs containing data and instructions used at the final dispatching steps. In addition, $APIC_1$ prepares the timer to generate a final interrupt at time $t_r - \delta$ and immediately switches the context into a dummy idle task, called "APIC idle". The idle task is carefully engineered to preserve the contents of the level-1 cache, thereby eliminating the side-effects of the non real-time task that are evident in the single interrupt approach. The final interrupt, $APIC_0$, performs the context switching back into the kernel, which in turn dispatches the real-time task. The values of τ and δ are empirically determined to accommodate the context-switching transitions 2, 3 and 4 and still minimize the average value of err_{Task} .

Appendix A contains excerpts from a paper providing a more detailed description of the research and results from the work described in this section.

3 Parametric Scheduling

A new design for the Maruti scheduling scheme has been developed to enhance the system task schedulability, and broaden the range of task types that can be scheduled by the system in a timely manner. The following is a description of the modules of the scheduling model, their scheme of execution, and information passing:

Off-line scheduler The off-line module accepts an ordered set of tasks along with their timing requirements such as ready time, deadline, period jitters, and relative timing constraints among the different tasks. It uses this information to generate a dynamic calendar.

Dynamic calendar The dynamic calendar contains information about the task instances and their timing dependencies in the form of functions whose parameters are values generated at run-time, such as internal system states, external system physical state, or previous task instances actual execution times. The parametric functions produce the minimum and maximum starting times for the different task instances as their output. The calendar also include a pointer to the first task instance to be executed.

On-line Dispatcher This module executes as part of the operating system kernel. It is initiated after an invocation request for the application is made. It starts by loading the dynamic calendar generated by the off-line module and uses the pointer in the dynamic calendar to dispatch the first task for execution. The on-line module remains active at run-time, filling in the values of the functions' parameters by values generated at run-time. It uses the times generated by these function to start the execution of the task instances according to their original timing constraints. The execution times of the different tasks are used as parameters for the functions of the parametric functions as well as a feedback for the off-line component, to be used as estimates for tasks execution times.

Using the new dynamic time-based scheduling scheme, we can schedule both periodic and aperiodic tasks. Tasks can execute in any general pattern other than strict periodic, for instance the system can schedule periodic tasks with variable inter-instance periods.

The dynamic time-based scheduling model can also support linear relative timing constraints that frequently rise in real-time applications. Linear relative timing constraints generally take the form $e_1 - e_2 \leq t$, where e_1, e_2 are timing events such as the start or finish of a task execution, and t is a time period.

The benefits of the scheduling scheme are:

- Ability to add aperiodic tasks at run-time.
- Ability to schedule more general tasks.
- Variation of the run-time behavior depending on values generated by executing tasks, or system state to change the parametric functions calculated by the off-line component at pre-runtime.
- Using parametric function makes use of the slack time to run non-real-time tasks, or to finish the schedule as early as feasible.

The proposed scheduling scheme also gives some possibilities of fault tolerance by allowing the operating system kernel to gain control, or update the

different functions parameters in case of failure. Some of the fault tolerance abilities that are supported by this scheme are:

- Substitution of minimum values for parameters in case of failure of task instances generating the parameter value to keep the feasible total schedule.
- Using the maximum execution time for the task instances to generate a time interrupt, should the task instance execute more than the max time allowed for it.

Appendix B contains excerpts from a paper providing a more detailed description of the research and results from the work described in this section.

4 MAGIC Tool

The MAGIC (Maruti Application Graphical Integration and Configuration) graphical development environment pulls together various aspects of system development, including compilation, configuration, scheduling, and analysis, and debugging in an integrated GUI framework.

MAGIC is designed to support various phases of the application development process. It accepts descriptions of tasks, which may be program modules. These modules are used as building blocks for the application. Magic supports a graphical manipulation of these tasks to establish their functional, timing and communication relationships. A framework for analyzing the resource usage is created which is used by the scheduler to generate the calendars for the application. The run time executable is created from the object code and the calendar. This executable can be run within MAGIC for debugging, including the analysis of temporal properties through temporal debugging. Once debugged, the executable generated by MAGIC can be used in the production Maruti-based designs. MAGIC has analysis tools during all phases of the development.

Following is a list of features for MAGIC:

- **Graphical view of the application structure.** Each component type and its role in the application are immediately evident in a graphical display of the topology. Figure 1 shows an example of such display.
- **Graphical view of the resources of the system.** Information about system resources available for use by the application can be graphically displayed in MAGIC, and back-end tools may be used to ascertain schedulability of the application.
- **Hierarchical grouping of components.** As a graphical representation of a large application may become very complex and unwieldy, MAGIC supports hierarchical grouping of components into blocks, and information

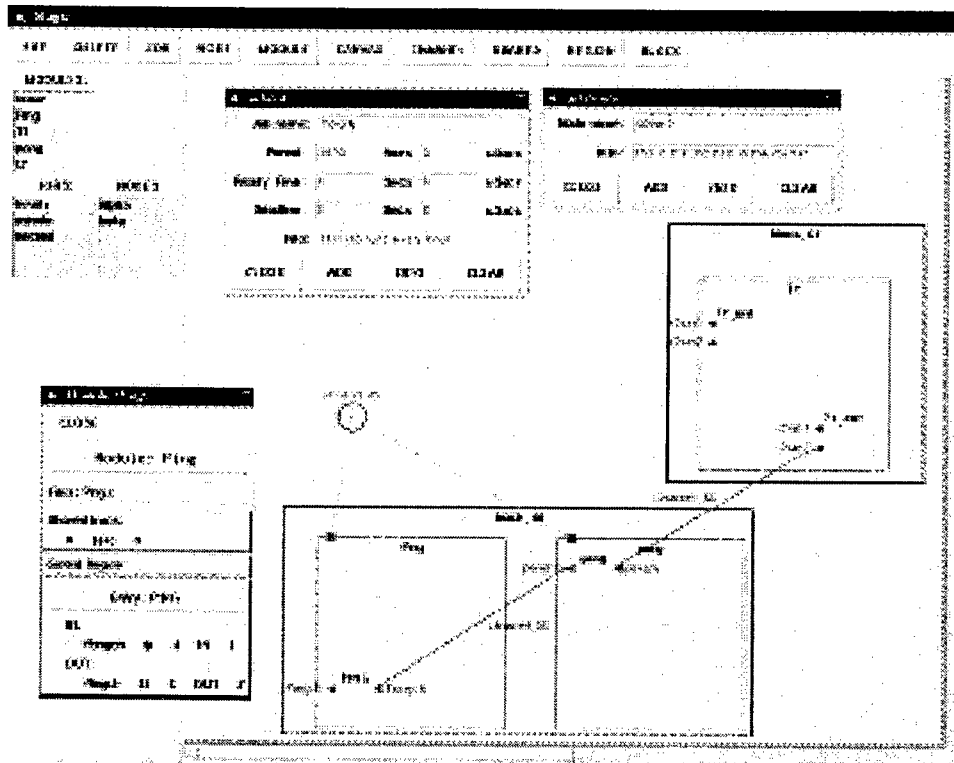


Figure 1: Screen Display of MAGIC

hiding in the display. For example, we may show only the interactions of a block with the outside world and not the structure of the block itself. Display filters facilitate viewing a selected part of the topology.

- **Multiple back-end analysis tools.** MAGIC supports incorporation of various analysis tools working from the common representation of the application structure and providing results which are available in the graphical format. Temporal analysis techniques will be added to the current repertoire of MAGIC.

We have finished a prototype implementation of MAGIC, which currently supports: the creation of tasks, threads, and jobs, the hooking up of communication channels, the transparent replication of tasks for fault-tolerance, and generation of Maruti executable. Continuing and ongoing works will incorporate the remaining features listed above into the common graphical framework of MAGIC.

A Dispatching Real-Time Tasks: A High-Precision Approach

In this chapter, we study the accuracy of two approaches to dispatch real-time tasks. We start by presenting the traditional approach of using count-down timers and discuss the factors affecting its precision. Next, we introduce a new approach for the deployment of the count-down timers that promises a higher degree of dispatching precision. For both approaches, we also accommodate the scheduling of non real-time tasks utilizing the CPU slack time.

A.1 The Single-Interrupt Approach

The typical approach for the deployment of count-down timers is to preset the timer with a specific value representing the number of clock ticks required to elapse before a timer interrupt is generated. Till that time, the CPU can be used to run non real-time tasks. On the arrival of the timer interrupt, the CPU launches an Interrupt Service Routine (ISR) which handles the context switching from the current task to the real-time task.

In our implementation, we make use of the count-down timer present in the Advanced Programmable Interrupt Controller (APIC) which is part of the Intel® P6 Architecture. The P6 architecture is implemented in all Pentium Pro® and newer Intel microprocessors. In these systems, the system bus typically runs at a clock cycle of 66.667 MHz¹. The CPU multiplies the bus clock to synthesis its internal CPU clock. For example, to run the CPU at a 300-MHz clock, the bus clock is multiplied by a factor of 4.5 to generate the desired CPU clock frequency.

The APIC timer has a 32-bit register that is decremented at each bus clock tick. For example, the counter is decremented every 15 nanoseconds when driven by a 66-MHz bus clock. The timer can be programmed to operate in one of two modes; “periodic”, or “one-shot”. In the periodic mode, the timer generates an indefinite number of interrupts periodically as determined by the initial value written in its register. Whereas in the one-shot mode, the timer generates only one interrupt.

Due to the limited width of the countdown register, the frequency of the bus clock, as well as other implementation-specific considerations, the longest period to program the timer may be limited to a maximum of M seconds. A typical value of M^2 is 1.431 seconds for a 300 MHz Pentium II driven by a 66 MHz bus clock. If a longer period, say T , is needed, the timer may be programmed in periodic mode for $\lfloor \frac{T}{M} \rfloor$ iterations of length M seconds each, followed by a one-shot iteration of length $(T \text{ modulo } M)$ seconds.

¹Faster bus clocks can now reach 100 MHz

²Please, refer to equation 1 for details on the derivation of M .

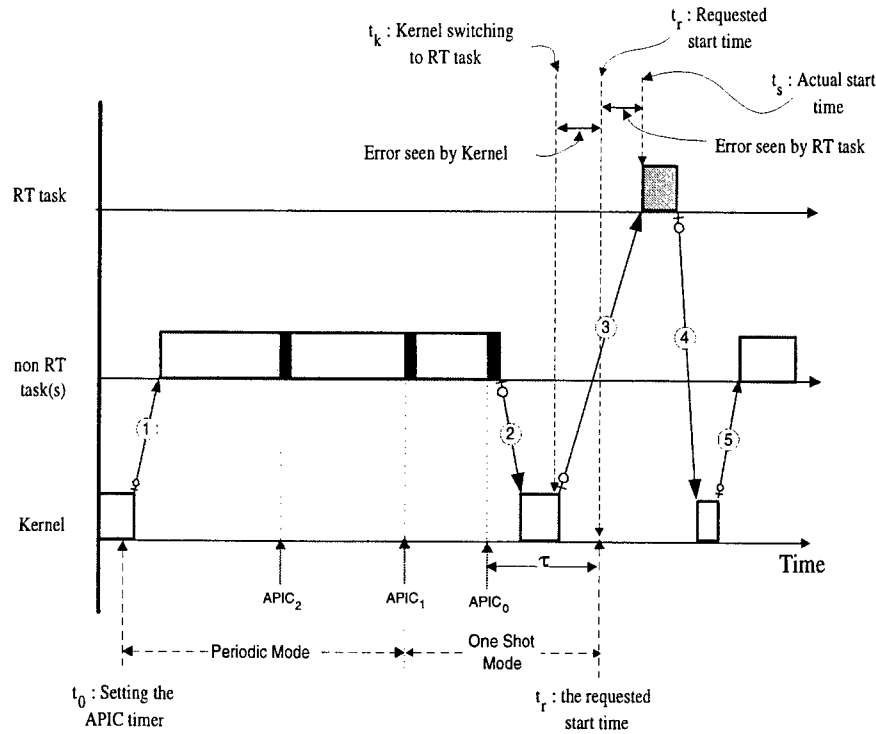


Figure 2: Using a single APIC timer interrupt to dispatch real-time tasks

This technique is further illustrated in figure 2. In this figure, which is not drawn to scale, we depict the time-line of the kernel, the non real-time and the real-time tasks. At time t_0 the real-time kernel sets up the APIC timer and its ISR to dispatch a real-time task at time t_r . The timer is first operated in periodic mode for several iterations denoted by $APIC_n, \dots, APIC_1$. Finally, a one-shot interrupt $APIC_0$ is generated at time $t_r - \tau$. The value of τ is empirically determined to accommodate the time needed to perform the context switching back to the kernel, and from there to the real-time task. The kernel typically performs some management steps before the real-time task is actually started. A lower bound on the interval $t_r - t_0$ is mandated by the amount of processing needed to set up the timer and perform the context switching transitions 1, 2, and 3.

A.2 Evaluation of the Single-Interrupt Approach

The single-interrupt approach is implemented on a Pentium II 300 MHz platform driven by a 66 MHz system bus clock with 64 MB of installed RAM. Evaluation

of this approach proceeds as follows:

- The APIC timer is programmed so that a real-time task is to be started at time t_r .
- The CPU context is switched to some non real-time task.
- The time instance t_k at which the kernel is about to dispatch the real-time task is recorded. The dispatching error as seen by the kernel, denoted by err_{kernel} , is defined to be $t_k - t_r$.
- The time instance t_s at which the real-time task is about to execute its first instruction is recorded. The dispatching error as seen by the real-time task itself, denoted by err_{task} is defined to be $t_s - t_r$.

The values of both err_{kernel} and err_{task} are monitored as the requested target time t_r is varied from few seconds down to few hundreds of nanoseconds. The non real-time task running in the background implements a recursive function that generates the Fibonacci series and store the results in a global data array. This selection increases the probability that the non real-time task will almost flush the contents of the processor's level-1 caches, both data and instruction caches, before the real-time task is dispatched. The results of a 4167-reading experiment are depicted in figure 3. This figure shows both err_{kernel} , which is the lower plot with the right-hand-side vertical axis, and err_{task} , which is the upper plot with the left-hand-side axis, versus the relative target time $t_r - t_0$ in units of CPU clock cycles³. It should be noted that when the experiment is repeated for the same set of t_r values, the same results are indeed repeatable. An inspection of figure 3 reveals the following:

- There is a lower bound on the value of $t_r - t_0$ mandated by the CPU time needed to execute the context switching transitions 1, 2, and 3 that are shown in figure 2. On our test platform, this lower bound was imperically found to be 666 CPU cycles (= 2.22 μ seconds).
- err_{kernel} varies within a range of 39 CPU cycles.
- err_{task} varies within a range of 65 CPU cycles.
- The similarity between the two curves suggests that the variability in err_{task} is a direct consequence of the variability in err_{kernel} , with the exception of very few instances. In fact, analysis of the difference $err_{task} - err_{kernel}$ shows that it behaves almost like a constant. This difference has a very small standard deviation equal to 0.6 CPU cycles.

³On a 300MHz system, 1 CPU clock cycle = $\frac{1}{4.5}$ bus clock cycles = $3\frac{1}{3}$ nanoseconds

- The increments in the values of both err_{kernel} and err_{task} are of the order of the main memory access cycle. This observation leads to the conclusion that the absence of data and/or instructions from the processor's level 1 cache is the main reason behind the variations in the dispatching time of the real-time task.

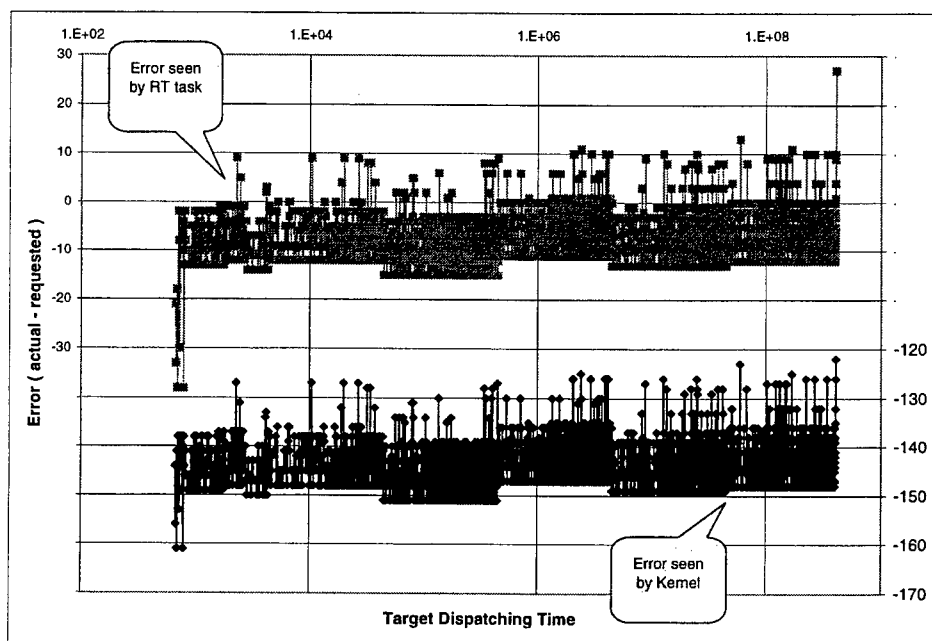


Figure 3: Dispatching time accuracy using single APIC interrupt

A.3 The Double-Interrupt Approach

The analysis of the single interrupt approach provided in section A.2 suggests that it is possible to reduce the amount of variability in the dispatching time through the absorption of the cache-miss effect. Pre-loading the CPU's level-1 cache with the required data and instructions has shown to significantly reduce the variability in the dispatching time. The technique that we use applies a double-interrupt approach to pre-load the data and instructions of that portion of the interrupt service routine responsible for the final dispatching procedures. Figure 4 illustrates this technique.

As with the single interrupt approach discussed in section A.1, the APIC timer may be programmed to generate few periodic interrupts, denoted by

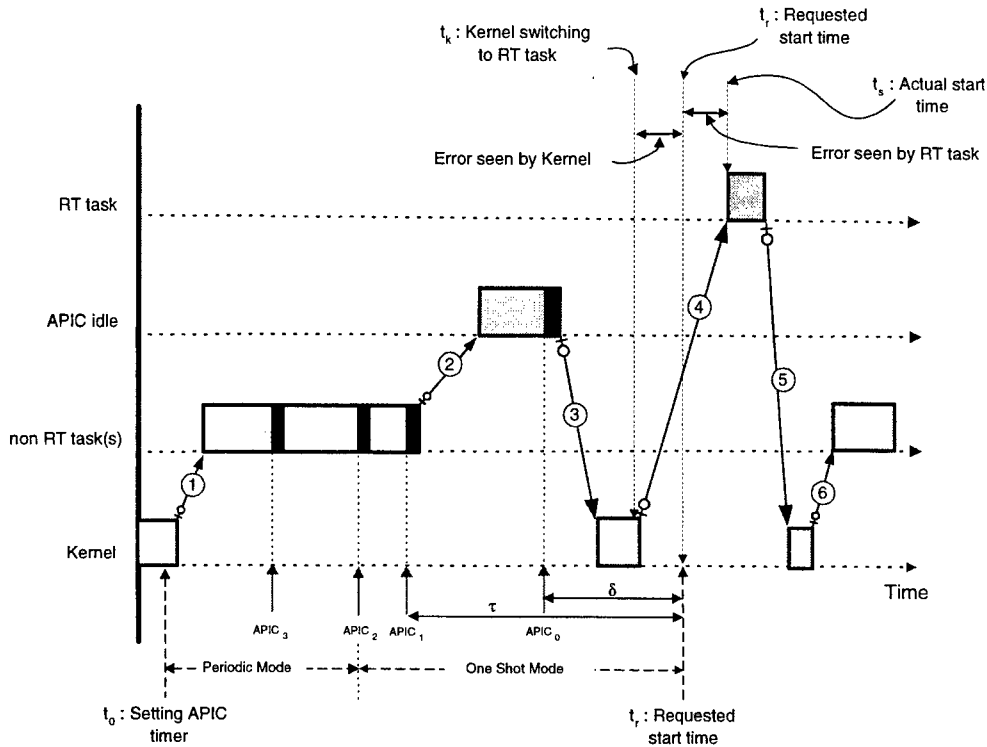


Figure 4: Using two APIC interrupts to dispatch real-time tasks

$APIC_n, \dots, APIC_2$. A semi-final interrupt, denoted by $APIC_1$, is generated at $t_r - \tau$ to pre-load the level-1 cache with the data and instructions required for the final dispatching steps. Fundamentally, $APIC_1$ touches all the memory paragraphs⁴ containing data and instructions used at the final dispatching steps. In addition, $APIC_1$ prepares the timer to generate a final interrupt at time $t_r - \delta$ and immediately switches the context into a dummy idle task, called "APIC idle" in figure 4. The idle task is carefully engineered to preserve the contents of the level-1 cache, thereby eliminating the side-effects of the non real-time task that are evident in the single interrupt approach. The final interrupt, $APIC_0$, performs the context switching back into the kernel, which in turn dispatches the real-time task. The values of τ and δ are empirically determined to accommodate the context-switching transitions 2, 3 and 4 and still minimize the average value of err_{Task}

⁴a memory paragraph is 32-byte long

A.4 Implementation of The Double-Interrupt Approach

The implementation of the double-interrupt approach involves two main components; namely

- a set-up routine to schedule an event at a specific time instance in the future, and
- the interrupt service routine (ISR) responding to the interrupts generated by the APIC timer.

The set-up routine is illustrated in figure 5. This routine receives three inputs:

1. The target time at which the event is to be dispatched
2. A pointer to the routine which implements the event
3. A pointer to a list of parameters to be passed to this specific instance of the event

The target time must not be sooner than a minimum threshold which is empirically determined to be 1200 CPU cycles. As it is mentioned in section A.1, the number of required periodic shots, if any, is $\left\lfloor \frac{t_r - t_0}{M} \right\rfloor$. The value of M is mandated by the computational details of the implementation and is given by:

$$M = \frac{2^{32}}{10 \times \text{CPU frequency}} \text{ seconds} \quad (1)$$

The total number of APIC interrupts, denoted by "cntDwn", equals 2 + number of periodic shots. The remaining time to event dispatching after all periodic shots have occurred, denoted by "cycles", is given by $(t_r - t_0) \bmod M$. Again, the value of "cycles" has to be more than the same threshold imposed on $(t_r - t_0)$. Finally, the value of "cntDwn" determines the programming mode of the APIC timer; either "one-shot" or "periodic" mode. In the periodic mode, the APIC timer is set up to generate multiple interrupts M seconds apart.

Next, the interrupt service routine (ISR) for the APIC timer is shown in figure 6. Each time an interrupt is received from the APIC timer, the "cntDwn" counter is decremented. As long as "cntDwn" is still greater than 2, the APIC timer continues to operate in periodic mode. The moment "cntDwn" becomes 2, a one-shot interrupt is scheduled τ ticks before the target time t_r . When this interrupt is received, "cntDwn" becomes 1, and the last interrupt is now scheduled δ ticks before the target time t_r . At this moment, the context is switched to the APIC Idle thread. As the last interrupt arrives, indicated by "cntDwn" becoming zero, the context is switched to the kernel in preparation for dispatching the real-time event.

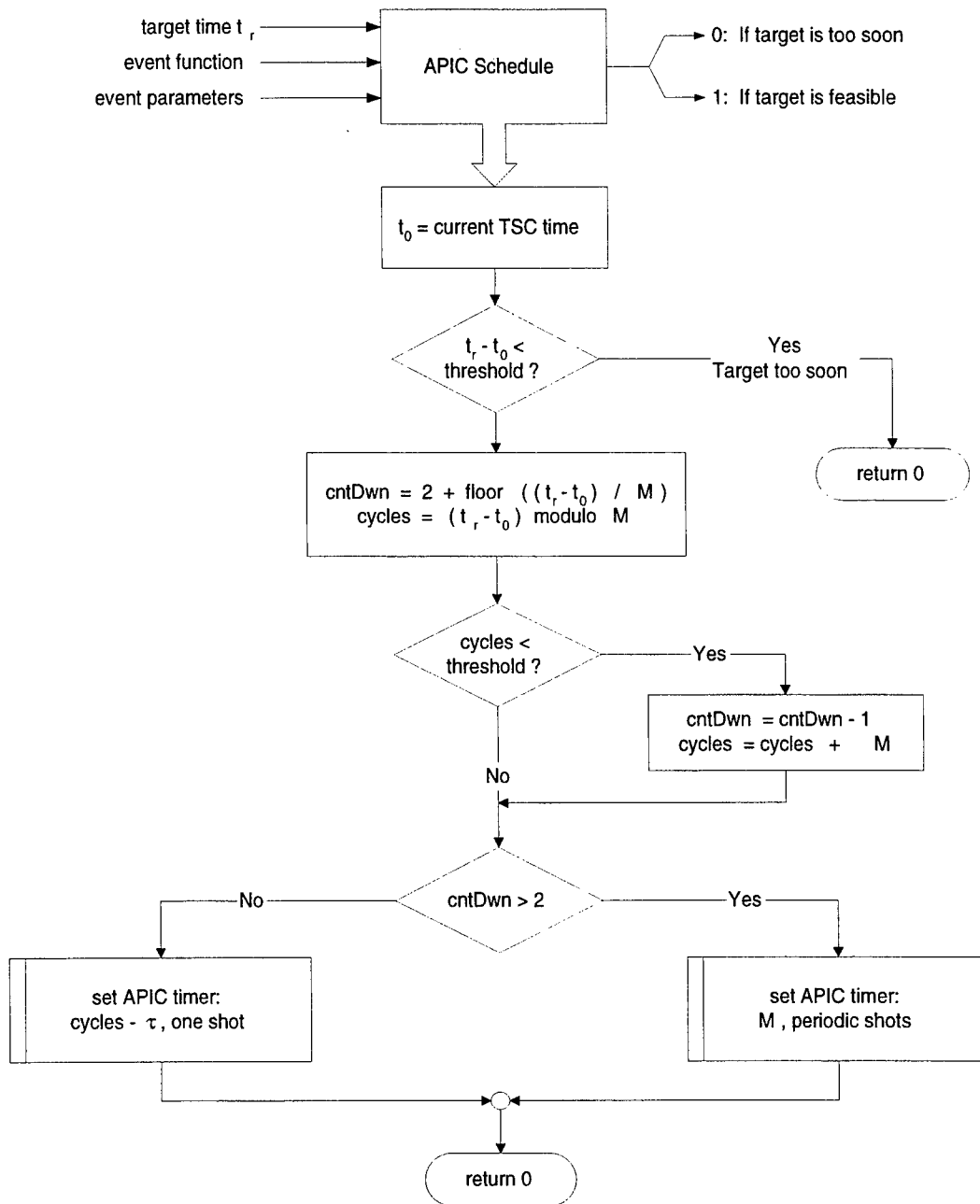


Figure 5: Setting up the APIC timer for double-interrupts

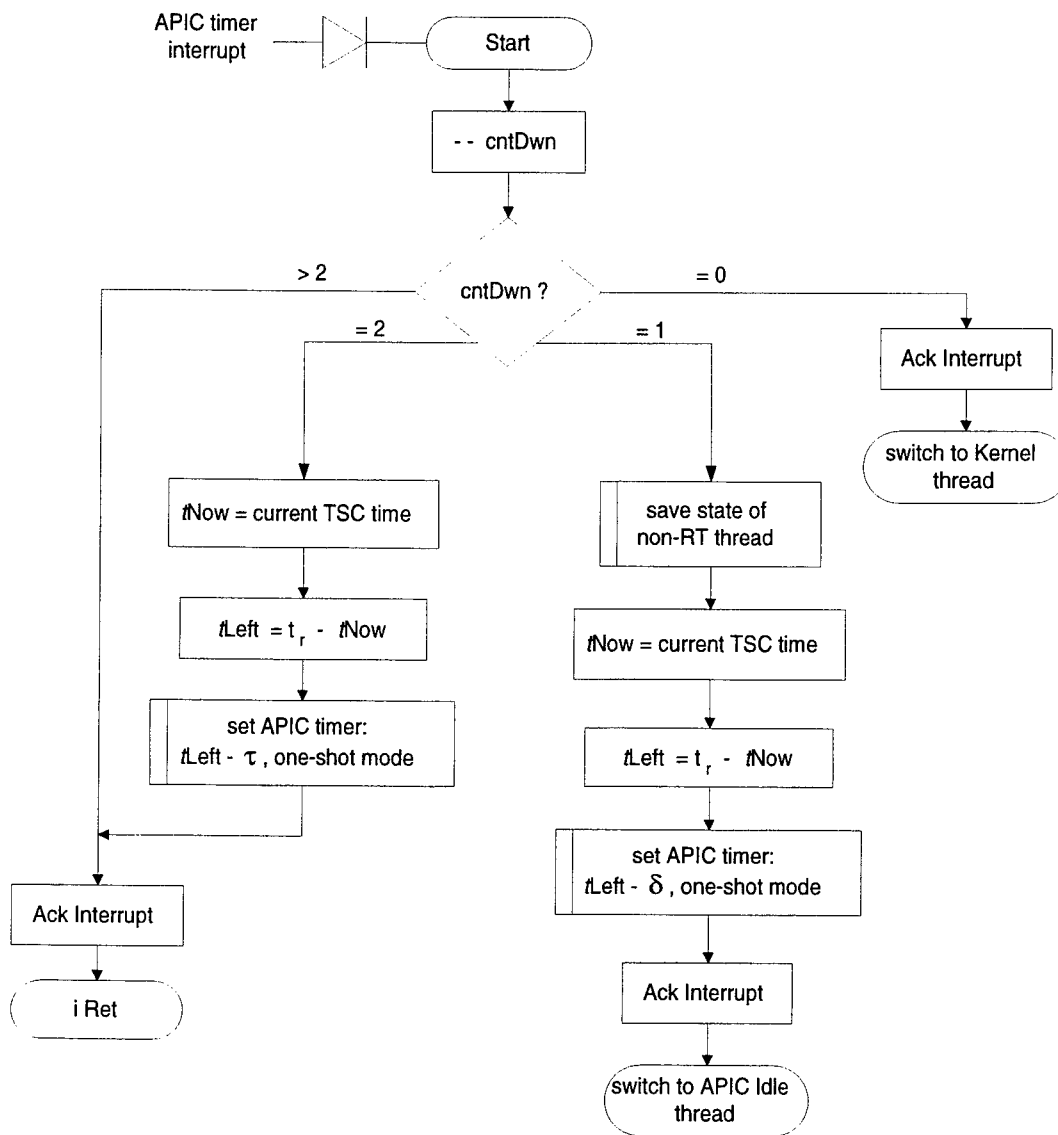


Figure 6: The interrupt service routine for the APIC timer

A.5 Evaluation of the Double-Interrupt Approach

The performance of the double-interrupt approach is evaluated using an experiment identical to that used for the single-interrupt approach in section A.2, with the same non real-time task in the background. The observed values of err_{kernel} and err_{task} are plotted in figure 7.

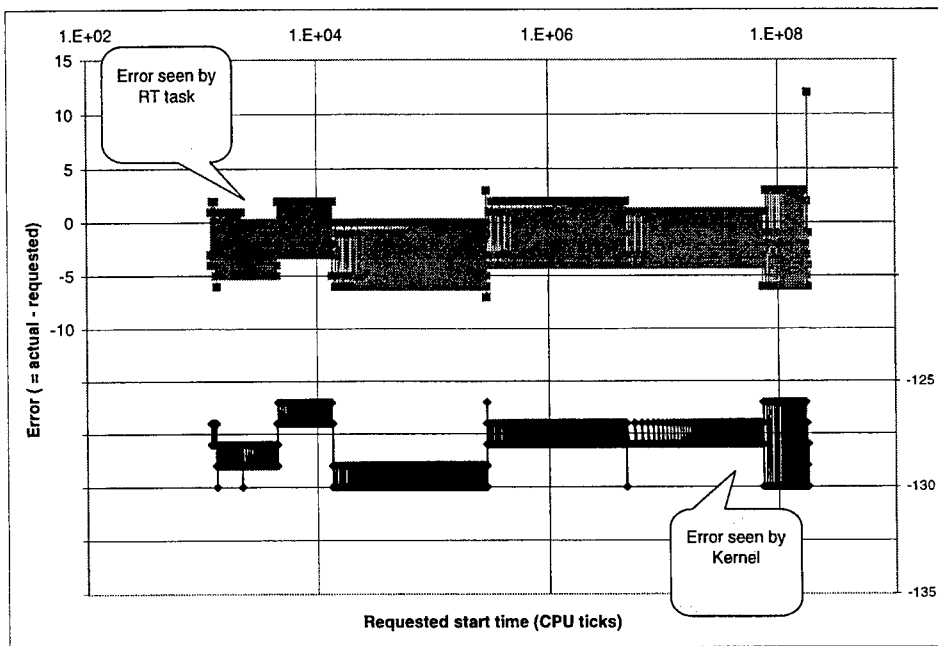


Figure 7: Accuracy of dispatching time using double APIC interrupts

An analysis of this figure reveals the following:

- The lower bound on the value of $t_r - t_0$ is 1200 CPU cycles ($=4\mu\text{Sec}$).
- err_{kernel} varies within a range of 4 CPU cycles.
- err_{task} varies within a range of 19 CPU cycles.

The lower bounds for τ and δ are empirically found to be 1125 and 490 CPU cycles, respectively⁵. Further reduction in these values, especially in the value of τ , resulted in erroneous kernel behavior.

⁵On our test platform, these are equivalent to 5.06 and 1.63 $\mu\text{seconds}$, respectively

A.6 Performance Comparison

Measure	Kernel Error			Task Error		
	Single Interrupt	Double Interrupts	% Improve	Single Interrupt	Double Interrupts	% Improve
Minimum	-161	-130	19	-38	-7	82
Maximum	-122	-126	3	27	12	56
Range	39	4	90	65	19	71
Average	-143.9	-128.3	11	-7.9	-2.6	67
StdDev	4.5	1.0	77	4.6	2.5	46

Table 1: Statistical comparison of err_{kernel} and err_{task}

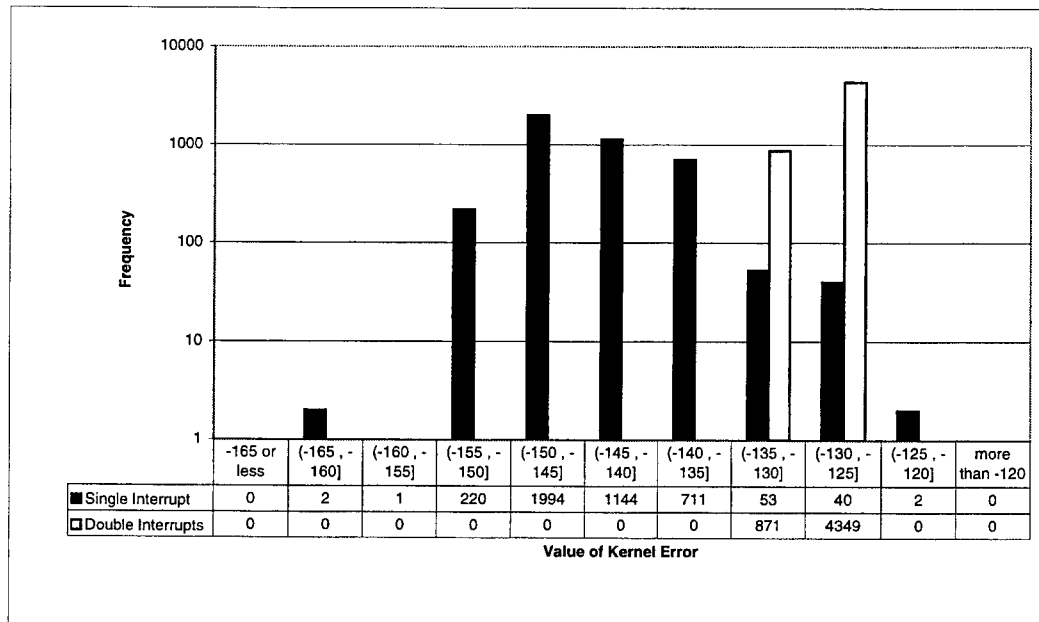


Figure 8: Frequency histogram of the err_{Kernel}

The task dispatching approaches presented in the previous sections provide two levels of temporal accuracy at two cost levels. Table 1 provides a side-by-side comparison of the single-interrupt and the double-interrupt approaches. The numbers indicate that the single interrupt dispatching mechanism results in a higher degree of uncertainty compared to the double-interrupt mechanism. The value of err_{kernel} has demonstrated a 90%-reduction in its variability range and

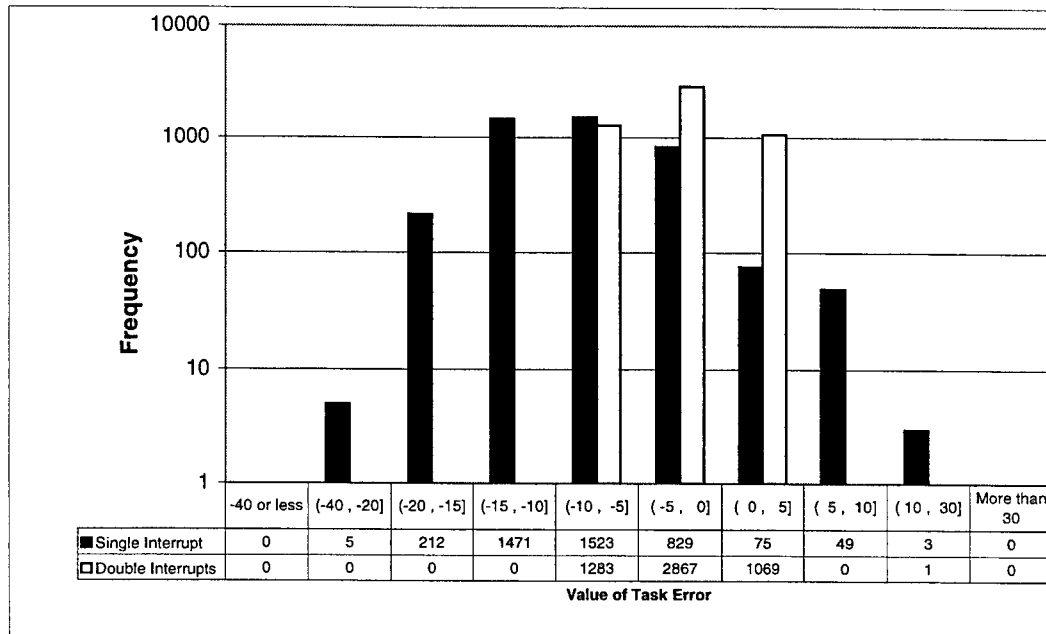


Figure 9: Frequency histogram of the err_{Task}

a 77%-reduction in its standard deviation when the double-interrupt technique is applied instead of the single-interrupt technique. Similarly, the value of err_{task} has shown 71% and 46% improvements in both statistics, respectively.

Figures 8 and 9 provide frequency histograms for err_{kernel} and err_{task} , respectively. These figures further illustrate the contribution of the double-interrupt dispatching approach in reducing the variability in the dispatching time.

On the other side, the increased accuracy has increased the overhead from 666 CPU cycles (2.22 μ seconds) to 1200 CPU cycles (4 μ seconds). We believe that for accuracy demanding hard real-time tasks, this overhead is a reasonable price to pay for the increased temporal accuracy.

A.7 Conclusion

In this chapter a new high-precision technique for dispatching real-time tasks is developed. The technique is based on a double-interrupt approach and works to counteract the temporal variability resulting from the CPU's level-1 data and instruction caches.

B Dynamic Time-Based Scheduling

In real time systems, relative timing constraints may be imposed on task executions, in addition to the release time and deadline constraints. Periodic tasks might also have jitter constraints between the start or finish times of any two consecutive executions [12].

This paper addresses the problem of scheduling and dispatching real-time tasks with inter-task temporal dependencies. An ordered set of N jobs is assumed to be given within a scheduling window and this ordering is cyclically repeated at runtime. An off-line scheduler is presented to check the schedulability of the job set and to obtain parametric lower and upper bound functions for the start times for the jobs, if the job set order is schedulable. An On-line dispatcher algorithm is given to evaluate these bounding functions at run-time to obtain a valid time intervals during which jobs can be started. These bounding intervals are used to order ready jobs in a time-ordered list for time-based dispatching [3].

The run-time execution timing model can be varied according to values generated by executing tasks, or system state to change the parametric functions calculated by the off-line scheduler at pre-runtime, and evaluated by the on-line dispatcher at run-time.

B.1 Introduction

Real-time systems are characterized by the presence of timing constraints on the computations carried out by the system. The timing constraints are statically determined at pre-runtime from the characteristics of physical systems they interact with. A special class of real-time systems, named *hard real-time systems*, require that the timing constraints be guaranteed prior to execution, since the result of a timing failure may lead to unstable or undesirable system behavior.

Many real-time systems are constructed in the form of a cyclic executive model in which the application tasks are dispatched according to a predetermined periodic schedule named the *calendar*. The calendar lists the tasks and their start times, or valid intervals to start the tasks executions. At run-time, the dispatcher uses the calendar to dispatch tasks at their pre-determined start times. This approach is particularly suitable for periodic activities which often constitute the major part of the load in many real-time systems.

While the problem of guaranteeing timing constraints in hard real-time systems has received significant attention, few techniques have addressed the problem of guaranteeing inter-task temporal dependencies such as relative timing constraints. Most real-time scheduling techniques consider the scheduling of real-time tasks with ready times and deadlines [4, 5, 16, 11, 14, 6, 8, 7]. These constraints impose constant intervals in which a task must be executed. In contrast, in the presence of relative time constraints, the time window within which a task must execute may depend on the scheduling of the other tasks in the sys-

tem. Some scheduling systems consider scheduling the problem of scheduling aperiodic tasks with relative timing constraints [9, 12].

In this section we briefly describe two scheduling schemes closely related to ours. The first one is the static cyclic scheduling scheme [2] and the second one is the parametric scheduling scheme [10].

B.1.1 Static cyclic scheduling

The static cyclic scheduling problem has been studied in [2]. The periodic task model is used, which means that every job has a release time and a deadline constraints, and only the jitter constraints between two job start times are allowed. An important assumption made in the work is that the start times of jobs in Γ^j are statically determined as offsets from the start of the j -th scheduling window $[(j-1)L, jL]$, and this schedule is invoked repeatedly by wrapping around the end point of the current schedule to the start point of the next. In other words, $s_i^{j+1} = s_i^j + L$ holds for all $1 \leq j$.

In the presence of jitter constraints, the job start times should be chosen carefully such that the jitter constraints are satisfied at run-time as well as the absolute constraints. Obtaining the ordering and job start times is an NP-hard problem, since non-preemptive scheduling problem with release time and deadline constraints is NP-hard. Several priority based non-preemptive scheduling algorithms are presented and their performances are compared in [2].

Suppose that a job $\tau_{i_1}^j$ belongs to Γ^j , and a job $\tau_{i_2}^{j+1}$ belongs to Γ^{j+1} , and they have jitter constraints $c_1 \leq s_{i_2}^{j+1} - s_{i_1}^j \leq c_2$ ($0 < c_1 \leq c_2 \leq L$). From the above assumption, $s_{i_2}^{j+1} = L + s_{i_2}^j$ holds. Thus, a new constraint is created, $c_1 - L \leq s_{i_2}^j - s_{i_1}^j \leq c_2 - L$, which is again equal to $L - c_2 \leq s_{i_1}^j - s_{i_2}^j \leq L - c_1$. Therefore, the jitter constraints across the boundary of Γ^j and Γ^{j+1} are transformed into jitter constraints between two jobs in Γ^j . As a consequence, if we can find a static schedule for Γ^j that satisfy the above transformed constraints and the constraints between jobs within Γ^j , it is clear that all timing constraints will be satisfied if that schedule is repeatedly used at run-time. This approach is depicted in Figure 10.

However, this approach suffers from the following limitations:

- The relative constraints allowed are limited to jitter type constraints between start times of two jobs.
- The schedulability of job sets are reduced due to the static start time assignments.
- It is very difficult to effectively incorporate dynamic tasks, such as aperiodic tasks, into a schedule by dynamically adjusting the start times of the jobs.

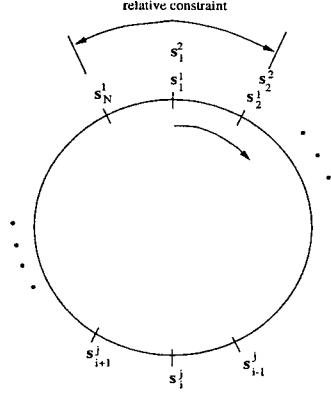


Figure 10: Static Cyclic Scheduling

In some real-time applications, the jitter constraints may be imposed between the finish times of the jobs rather than between the start times [1]. Furthermore, a periodic task may be decomposed into several subtasks and any kind of standard constraints may be defined between these subtasks [10]. In these cases this static scheduling approach is no more applicable without sacrificing the schedulability [10].

By transforming the jitter constraints across the boundary of Γ^j and Γ^{j+1} into those between jobs within Γ^j , we are affecting the schedulability of job sets. We will show that, under our new scheduling scheme in which this transformation is not necessary, the schedulability of job sets is increased, i.e., some job sets are not schedulable according to this scheme whereas it is schedulable by our scheme.

B.1.2 Parametric scheduling

Gerber *et al.* [10] proposes a parametric scheduling scheme in the scope of transaction scheduling, in which any standard constraints may be given between jobs in one transaction. Let $\Pi = \langle \tau_1, \dots, \tau_N \rangle$ denote a sequence of jobs constituting one transaction with a set of standard constraints, \mathcal{C} . Then, a schedulability of Π is defined as follows:

$$Sched \equiv \exists s_1 :: \forall e_1 \in [l_1, u_1] :: \dots :: \exists s_N :: \forall e_N \in [l_N, u_N] :: \mathcal{C} \quad (2)$$

From this *Sched* predicate, parametric lower and upper bound functions for each start time s_i are obtained by eliminating the variables in an order e_N, s_N, \dots, e_i . The parametric lower and upper bound functions, denoted as $\mathcal{F}_{s_i}^{min}$ and $\mathcal{F}_{s_i}^{max}$, are parameterized in terms of the runtime variables, $s_1, e_1, \dots, s_{i-1}, e_{i-1}$ of already executed jobs. The parametric calendar structure is shown in figure 11.

$\mathcal{F}_{s_1}^{min}()$	\leq	s_1	\leq	$\mathcal{F}_{s_1}^{max}()$
$\mathcal{F}_{s_2}^{min}(s_1, e_1)$	\leq	s_2	\leq	$\mathcal{F}_{s_2}^{max}(s_1, e_1)$
\vdots		\vdots		
$\mathcal{F}_{s_N}^{min}(s_1, e_1, s_2, e_2, \dots, s_{N-1}, e_{N-1})$	\leq	s_N	\leq	$\mathcal{F}_{s_N}^{max}(s_1, e_1, s_2, e_2, \dots, s_{N-1}, e_{N-1})$

Figure 11: Parametric Calendar Structure

This parametric calendar is obtained from an off-line component of the algorithm by applying variable elimination techniques that will be given later in this section, and the actual bounds of s_i are found at runtime by evaluating the parametric functions in the parametric calendar by using the start times and the finish times of already executed jobs, $\tau_1, \dots, \tau_{i-1}$. The actual form of these parametric functions are given in the following proposition.

Proposition 1 (Parametric Bound Functions [10]) *A parametric lower bound function for s_j is of the following form:*

$$\begin{aligned} & \mathcal{F}_{s_j}^{min}(s_1, f_1, \dots, s_{j-1}, f_{j-1}) \\ &= \max(p_1 + c_1, p_2 + c_2, \dots, p_a + c_a, \alpha_j^{min}) \end{aligned} \quad (3)$$

where each p_i , $1 \leq i \leq a$, belongs to $\{s_1, f_1, \dots, s_{j-1}, f_{j-1}\}$, and c_i is an arbitrary constant.⁶ And, α_j^{max} is a non-negative integer.

Similarly, a parametric upper bound function for s_j is of the following form:

$$\begin{aligned} & \mathcal{F}_{s_j}^{max}(s_1, f_1, \dots, s_{j-1}, f_{j-1}) \\ &= \min(q_1 + d_1, q_2 + d_2, \dots, q_b + d_b, \alpha_j^{max}) \end{aligned} \quad (4)$$

where each q_i , $1 \leq i \leq b$, belongs to $\{s_1, f_1, \dots, s_{j-1}, f_{j-1}\}$, and d_i is an arbitrary constant..

The main result obtained by the paper is that, for an arbitrary set of standard constraints on $\Pi = \{\tau_1, \dots, \tau_N\}$, we can find the parametric calendar in $O(N^3)$ time and the run-time evaluation of each bound function can be carried out in $O(N)$ time.

By applying this parametric scheduling scheme, we are not only able to schedule any sequence of jobs with standard constraints, but also able to take advantage of the flexibility offered by the scheme. That is, the job start times may be decided dynamically at runtime to incorporate other dynamic activities in the system. Even though this scheme is directly applicable to our k -fold cyclically constrained job sets, if the number of jobs in $\Gamma^{1,k}$ becomes large, the bounds need to be found on the size of parametric functions and for the memory requirements for them. The Process of variable elimination is discribed in [12].

⁶Note that $f_i = s_i + e_i$.

B.2 Maruti Programming Environment

Building a Maruti application is a process that consists of a set phases, beginning with the coding of the Maruti programming language (MPL) which is used to develop individual program modules, and Maruti configuration language (MCL) which is used to specify how individual program modules are to be connected together to form an application and the details of the hardware platform on which the application is to be executed [15]. The life-cycle of an application in the Maruti environment can be divided into following phases Figure 12.

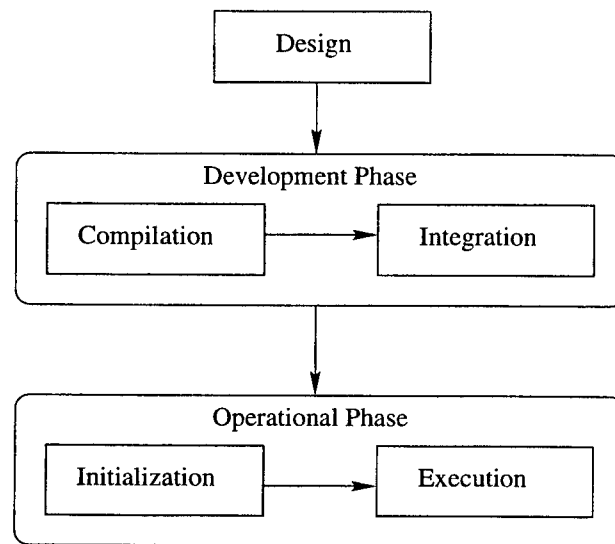


Figure 12: Maruti Application Life Phases

B.2.1 Design Phase

This stage is the starting point of the development of an application during which the overall design is carried out. The tasks layout and their timing requirements are specified [13].

B.2.2 Development Phase

This phase is broken down into two stages, namely compilation, and integration.

- *Compilation.* The source code modules created at this stage, along with their interface specifications. The resource requirements and the relative timing constraints for the modules are identified at this level, and are supplied to the integration environment.

- *Integration.* In this stage, modules created in the compilation stage are interconnected to form a complete program. The resource requirements for the application are identified and recorded with the application. The timing characteristics of the application is captured in the form of parametric functions that are passed to the dispatcher to be evaluated at run time.

The result of this phase is an executable application program, along with its resource and timing requirements [13].

B.2.3 Operation Phase

In this phase resource allocation, dispatching, execution for the tasks occur. It is initiated after an invocation request is made. This phase consists of two stages, dispatcher initialization and task execution.

- *Dispatcher initialization.* The Maruti on-line dispatcher loads the parametric functions for the invoked application tasks, and constructs the dynamic calendar that is used to dispatch the tasks in a timely manner.
- *Execution.* During this stage the operating system kernel performs dispatching, message passing, and reservation enforcement. Previous stages prepare the application for this stage, so that the timing requirements for the application are met, and the run-time overheads are minimal. The actual running time for each task instance is recorded as a form of feedback for the off-line scheduler to be used in generating more accurate calendars.

The resource requirements and timing information are identified and tracked as the application progresses through its life cycle, and are explicitly used during run time [13].

B.3 Model Description

This section describe the different modules of the scheduling-execution model and their scheme of execution and information passing Figure 13. The dynamic time-based scheduler consists of the following modules

B.3.1 Off-line scheduler

The off-line module accepts an ordered set of tasks along with their timing requirements such as ready time, deadline, period jitters, and relative timing constraints among the different tasks. It uses this information to generate a dynamic calendar.

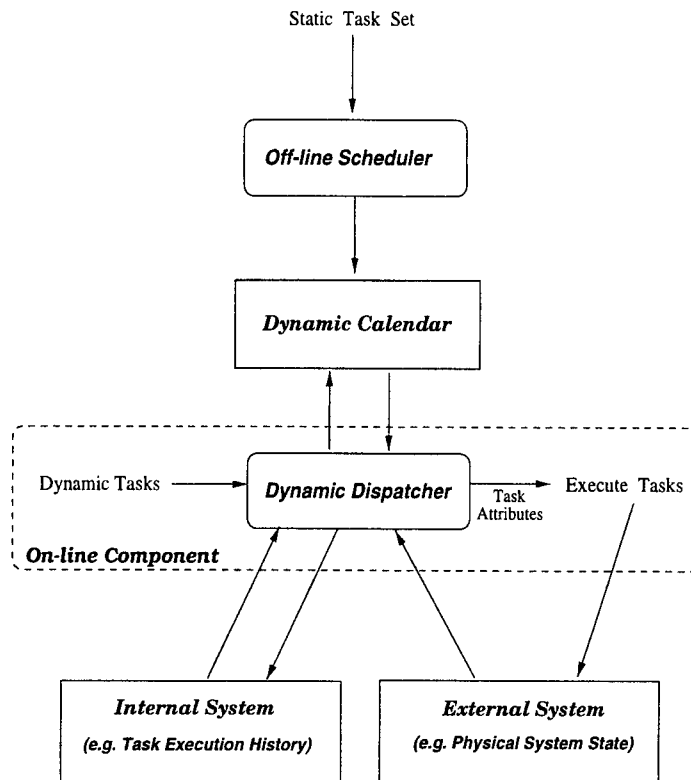


Figure 13: Operation Model of the dynamic time-based scheduling system

B.3.2 Dynamic calendar

The dynamic calendar contains information about the task instances and their timing dependencies in the form of functions whose parameters are values generated at run-time, such as internal system states, external system physical state, or previous task instances actual execution times. The parametric functions produce the minimum and maximum starting times for the different task instances as their output. The calendar also include a pointer to the first task instance to be executed.

B.3.3 On-line Dispatcher

This module executes as part of the operating system kernel. It is initiated after an invocation request for the application is made. It starts by loading the dynamic calendar generated by the off-line module and uses the pointer in the dynamic calendar to dispatch the first task for execution. The on-line module remains active at run-time, filling in the values of the functions' parameters

by values generated at run-time. It uses the times generated by these function to start the execution of the task instances according to their original timing constraints. The execution times of the different tasks are used as parameters for the functions of the parametric functions as well as a feedback for the off-line component, to be used as estimates for tasks execution times.

B.4 Off-line Scheduler

As in the parametric scheduling approach developed for transaction scheduling [10], we want to devise a schedulability test and an efficient dispatching mechanism when an ∞ -fold cyclically constrained job set, $\Gamma^{1,\infty}$, is given with its constraint matrices and vectors. We say $\Gamma^{1,k}$, is *schedulable* if there exists any method which can successfully dispatch the jobs in $\Gamma^{1,k}$.

Definition 1 (Schedulability of $\Gamma^{1,k}$) *The k -fold cyclically constrained job set $\Gamma^{1,k}$ ($1 \leq k$) is schedulable if the following predicate holds:*

$$\begin{aligned} sched^{1,k} \equiv & \exists s_1^1 :: \forall e_1^1 \in [l_1^1, u_1^1] :: \exists s_2^1 :: \forall e_2^1 \in [l_2^1, u_2^1] :: \dots \\ & \exists s_N^k :: \forall e_N^k \in [l_N^k, u_N^k] :: \mathcal{C}^{1,k} \end{aligned} \quad (5)$$

where $\mathcal{C}^{1,k}$ is a set of standard constraints defined on $\{s_1^1, e_1^1, \dots, s_N^k, e_N^k\}$.

Then, the following proposition holds for all $k \geq 1$.

Proposition 2

$$\forall k \geq 1 :: sched^{1,k+1} \implies sched^{1,k}$$

Proof: Obvious from the definition of a cyclically constrained job set and from the definition of $sched^{1,k}$ in (5).

Hence, once $sched^{1,k}$ turns out to be **False**, then all $sched^{1,j}$, $k \leq j$, are **False**, too. By this proposition, the schedulability of $\Gamma^{1,\infty}$ is defined.

Definition 2 (Schedulability of $\Gamma^{1,\infty}$) $\Gamma^{1,\infty}$ is schedulable if and only if

$$\lim_{k \rightarrow \infty} sched^{1,k} = \mathbf{True}$$

In [10], it is shown that checking Predicate (2) is not trivial because of the nondeterministic job execution times and because of the existence of standard relative constraints among the jobs. This applies to the above $sched^{1,k}$ predicate, too. The variable elimination techniques are used in [10] to eliminate variables from Predicate (2). At the end of the variable elimination process parametric bound functions for s_i , that are parameterized in terms of the variables in $\{s_1, e_1, \dots, e_{i-1}\}$, are found as well as the predicate value.

However, if we want to apply the variable elimination technique to $sched^{1,k}$, the following problems have to be addressed first:

1. On which subset of $\{s_1^1, e_1^1, \dots, s_{i-1}^j, e_{i-1}^j\}$ does the parametric bound functions for s_i^j depend?
2. Is it required to store parametric bound functions for every job in $\Gamma^{1,k}$?
3. What parametric bound functions have to be used if k is not known at pre-runtime and dynamically decided at runtime?

Let $\mathcal{F}_{s_i^j}^{min,k}$ and $\mathcal{F}_{s_i^j}^{max,k}$ denote parametric lower and upper bound functions for s_i^j , respectively, that are found after the variable elimination algorithms are applied to $sched^{1,k}$. If the number of variables is unbounded with which $\mathcal{F}_{s_i^j}^{min,k}$ or $\mathcal{F}_{s_i^j}^{max,k}$ is parameterized, then it is not possible to evaluate them at runtime within bounded computation times. Also, if it is required that parametric bound functions for every job in $\Gamma^{1,k}$ be stored at runtime, the scheme is not implementable for large k because of memory requirements. Finally, if the value of k is not known at pre-runtime and is decided dynamically at runtime, which is true in most real-time applications, the parametric bound functions to be used have to be selected.

In this section, the answers to the above questions are sought by first transforming $sched^{1,k}$ into a constraint graph and by investigating the properties of such graphs.

B.4.1 Transforming a constraint set into a constraint graph

We want to apply the variable elimination algorithms to $sched^{1,k}$ for some fixed k , and want to find out answers to the previously raised three questions. For that purpose, we first transform the predicate into a constraint graph and apply node elimination algorithms (corresponding to the variable elimination algorithms) to the graph. Then, the properties of the constraint graphs created during the node elimination process are examined. Working on constraint graphs, instead of constraint sets themselves, makes it easier to infer and prove useful properties. In this section, the transformation rules are given for a set of jobs and its associated constraint set.

Let $\Pi = \{\tau_1, \tau_2, \dots, \tau_N\}$ be a finite set of jobs with a set of standard constraints, \mathcal{C} . Consider eliminating quantified variables from the following predicate:

$$Sched \equiv \exists s_1 :: \forall e_1 \in [l_1, u_1] :: \dots \exists s_N :: \forall e_N \in [l_N, u_N] :: \mathcal{C}$$

Then, predicates on subsets of $\{s_1, e_1, \dots, s_N, e_N\}$ are defined next that are found after eliminating variables.

Definition 3 *Sched(s_a) ($1 \leq a \leq N$) is defined to be a predicate on a set of variables $\{s_1, e_1, \dots, s_a\}$ that are found after eliminating variables of $\langle f_N, s_N, \dots, f_a \rangle$ from Sched. Sched(e_a) is defined similarly.*

That is, $Sched(s_a)$ can be expressed as

$$Sched(s_a) \equiv \exists s_1 :: \forall e_1 \in [l_1, u_1] :: \dots \exists s_a :: C(s_a)$$

It will be shown that $Sched$ (or $Sched(s_a)$, or $Sched(e_a)$) can be transformed into a directed graph, which is called a *constraint graph*, such that the variable elimination process can be mapped into a corresponding node elimination operation in the graph. Note that, in the following definition of a constraint graph, *semi-exclusive-ORed* edges are defined. Also, $v_1 \xrightarrow{w} v_2$ denotes an edge from a node v_1 to a node v_2 with a weight w , and $\langle v_1 \xrightarrow{w_1} v_2 \xrightarrow{w_2} \dots \xrightarrow{w_{i-1}} v_i \rangle$ denotes a path from a node v_1 to a node v_i with a weight sum $w = \sum_{j=1}^{i-1} w_j$. $v_1 \rightsquigarrow v_i$ denotes that there exists a path from v_1 to v_i , and $v_1 \overset{w}{\rightsquigarrow} v_i$ denotes that there exists a path from v_1 to v_i whose weight sum is w .

The following rule is used to transform a predicate into a constraint graph.

Definition 4 (Constraint Graph) A constraint graph $G(V, E)$ is found from $Sched$ (or $Sched(s_a)$, or $Sched(e_a)$) as follows:

1. node set V is obtained as follows:

- $v_0 \in V$
- $s_i, f_i \in V$ for $1 \leq i \leq N$ where $f_i = s_i + e_i$.

2. edge set E is obtained as follows:

- For each tuple $\langle s_i, f_i \rangle$, add the following semi-exclusive-ORed edges to E :
 - (a) $s_i \xrightarrow{l_i} f_i$
 - (b) $f_i \xrightarrow{-u_i} s_i$
- For each constraint in C that can be converted to:
 - (a) $v_i - v_j \leq c$ ($v_i, v_j \in \{s_i, f_i \mid 1 \leq i \leq N\}$): add $v_j \xrightarrow{c} v_i$ to E .
 - (b) $v_i \leq c$: add $v_0 \xrightarrow{c} v_i$ to E .
 - (c) $-v_i \leq c$: add $v_i \xrightarrow{c} v_0$ to E .

Definition 5 The constraint graph found from $Sched(s_a)$ is denoted as $G(s_a)$.⁷ Similarly, $G(f_a)$ represents a graph found from $Sched(e_a)$.

Figure 14 shows a graph created from an example job set $\Gamma^{1,2}$. Note that v_0 is an extra node created to represent a constant 0 that is used to specify absolute constraints such as the release time and the deadline constraints. In the figure, the edges connected by \oplus are semi-exclusive-ORed edges.

⁷The full notation would be $G(s_a)(V, E)$. But, if no confusion is caused, $G(s_a)$ will be used in this chapter.

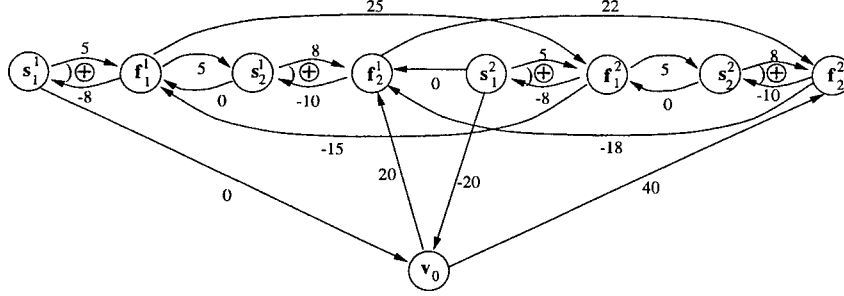


Figure 14: Constraint Graph for $\Gamma^{1,2}$

Note that there may exist only one edge from one node to another from the uniqueness of inequality in the constraint set. For example, if there are two constraints $v_1 - v_2 \leq c_1$ and $v_1 - v_2 \leq c_2$ in \mathcal{C} , then one of them is redundant. Therefore, we can denote an edge from v_1 to v_2 in a constraint graph as $v_1 \rightarrow v_2$ without its weight specified. Also, note that any edge from f_i to s_i is semi-exclusive-ORed to any edge from s_i to f_i . That is, even if any of these two edges is created from another constraint in \mathcal{C} rather than from the minimum or maximum execution time constraint, they are semi-exclusive-ORed.

The elimination algorithm of a node f_a from a graph $G(f_a)$ is presented next.

Algorithm 1 (Elimination of f_a from a Graph $G(f_a)$) *Elimination of f_a from $G(f_a)$ is performed by the following algorithm.*

1. For each edge pair, $\langle y \xrightarrow{w_1} f_a, f_a \xrightarrow{w_2} s_a \rangle$, that are not semi-exclusive-ORed in $G(f_a)$:

- create an edge $y \xrightarrow{w_1+w_2} s_a$.
 - (a) If $y = s_a$ and $w_1 + w_2 < 0$, then return **False**.⁸
 - (b) If $y = s_a$ and $w_1 + w_2 \geq 0$, then remove this edge.⁹
 - (c) If there already exists an edge $y \xrightarrow{w'} s_a$ before creating $y \xrightarrow{w_1+w_2} s_a$, then the edge with less weight remains, while the other is removed.

2. For each edge pair, $\langle s_a \xrightarrow{w_1} f_a, f_a \xrightarrow{w_2} z \rangle$, $z \neq s_a$, that are not semi-exclusive-ORed in $G(f_a)$:

- create an edge $s_a \xrightarrow{w_1+w_2} z$.

⁸This is because $y - y = 0 \leq w_1 + w_2 < 0$ is a contradiction.

⁹This is because $y - y = 0 \leq w_1 + w_2$ is a tautology.

- (a) If there already exists an edge $s_a \xrightarrow{w''} z$ before creating $s_a \xrightarrow{w_1+w_2} z$, then the edge with less weight remains, while the other is removed.

3. Set $V = V - \{f_a\}$ and remove all edges to or from f_a in $G(f_a)$.

Let $Elim(G(f_a), f_a)$ denote a new graph created after eliminating f_a from the graph $G(f_a)$ according to Algorithm 1 in case **False** is not found. In [3] it is shown that $Elim(G(f_a), f_a)$ is equivalent to the original graph $G(f_a)$.

Next, we show how a node corresponding to an existential quantifier s_a may be eliminated from the graph $G(s_a)$.

Algorithm 2 (Elimination of s_a from a Graph $G(s_a)$) Elimination of s_a from $G(s_a)$ is performed by the following algorithm.

1. For each edge pair, $\langle y \xrightarrow{w_1} s_a, s_a \xrightarrow{w_2} z \rangle$, in $G(s_a)$:
 - create an edge $y \xrightarrow{w_1+w_2} z$.
 - (a) If $y = z$ and $w_1 + w_2 < 0$, then return **False**.
 - (b) If $y = z$ and $w_1 + w_2 \geq 0$, then remove this edge.
 - (c) If there already exists an edge $y \xrightarrow{w'} z$ before creating $y \xrightarrow{w_1+w_2} z$, then the edge with less weight remains, while the other is removed.
2. Set $V = V - \{s_a\}$ and remove all edges to or from s_a in $G(s_a)$.

Similarly, let $Elim(G(s_a), s_a)$ denote a new graph created after eliminating s_a from the graph $G(s_a)$ according to Algorithm 2 in case **False** is not found. In [3] it is shown that $Elim(G(s_a), s_a)$ is equivalent to the original graph $G(s_a)$.

The elimination process of nodes, f_a and s_a , from the graph $G(f_a)$ can be viewed as preserving the connectivity between any two nodes in $\{v_0, s_1, f_1, \dots, s_{a-1}, f_{a-1}\}$ through f_a and s_a in $G(f_a)$. That is, if there exists any path from y to z only through s_a and f_a in $G(f_a)$, then a new edge from y to z is created to maintain the connectivity from y to z even after f_a and s_a are eliminated.

Figure 15 shows a graph and its node elimination processes for $sched^{1,2}$ that is derived from $\Gamma^{1,2}$.

The necessary condition for *Sched* to be true is the existence of a negative weight cycle in the constraint graph [3].

B.4.2 Off-line component

In this section, a $4N$ -node graph, called *basis graph*, is obtained to which we can cyclically apply variable elimination algorithm without explicitly obtaining

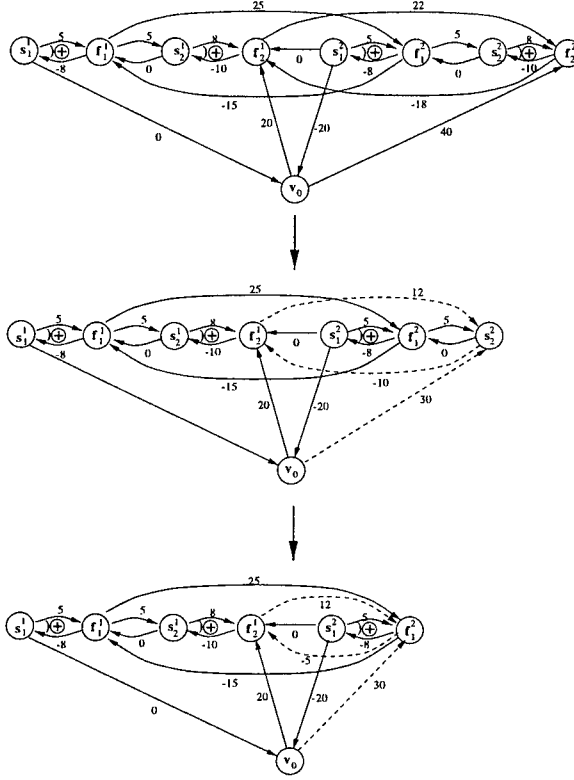


Figure 15: Elimination of f_2^2 and s_2^2 from $\Gamma^{1,2}$

a large constraint graph $G^{1,k}(f_N^k)$ for large k . That is, by recursively applying variable elimination algorithm to this smaller graph, it can be decided whether the created edge set sequence, $\Psi^{1,k}(f_N^j)$, $j = k, k-1, \dots$, will converge or not.

Definition 6 (Basis Graph) A basis graph $G_b(V_b, E_b)$ is defined as a subgraph of $G^{1,2}(f_N^2)$ as follows.¹⁰

1. $V_b = V_{b,1} \cup V_{b,2} \cup \{v_0\}$ where:

$$V_{b,2} = \{s_1^2, f_1^2, \dots, s_N^2, f_N^2\}$$

$$V_{b,1} = \{v \mid \exists (v \rightarrow u \vee u \rightarrow v) \in G^{1,2}(f_N^2) \wedge u \in V_{b,2} \wedge v \neq v_0\}$$

2. All edges in $G^{1,2}(f_N^2)$ connecting any two nodes in V_b are included into E_b .

¹⁰ $G^{1,2}(f_N^2)$ is found from $\Gamma^{1,2}$.

Then, the variable elimination process for a graph $G^{1,k}(f_N^k)$ can be transformed into an equivalent one by using a basis graph as follows:

Algorithm 3 *Cyclic algorithm to obtain $G^{1,k}(f_N^2)$.*

- *Input:* k , Basis Graph $G_b(V_b, E_b)$
- *Output:* $G^{1,k}(f_N^2)$

1. Initialize $i = 1$.
2. Initialize $G_{in}^1(V_b, E_{in}^1) = G_b(V_b, E_b)$.
3. From $i = 1$ to $i = k - 2$ repeat the following:
 - (a) Eliminate, from $G_{in}^i(V_b, E_{in}^i)$, the nodes of $V_{b,2}$ by alternately using Algorithm 1 and 2.
 - (b) If **False** is returned from Algorithm 1 or 2, then return **False**.
 - (c) Let $G_{out}^i(V_{b,1} \cup \{v_0\}, E_{out}^i)$ denote the resulting graph.
 - (d) If $i \geq 2$ and $G_{out}^i(V_{b,1} \cup \{v_0\}, E_{out}^i) = G_{out}^{i-1}(V_{b,1} \cup \{v_0\}, E_{out}^{i-1})$, then return $G_{in}^i(V_b, E_{in}^i)$.
 - (e) Let $G_{in}^{i+1}(V_b, E_{in}^{i+1}) = G_b(V_b, E_b)$
 - (f) For each edge $v_1 \xrightarrow{w_{12}} v_2$ in $G_{out}^i(V_{b,1} \cup \{v_0\}, E_{out}^i)$,
 - i. If $v_1 \neq v_0$ and $v_2 \neq v_0$, add an edge $g_{(1)}(v_1) \xrightarrow{w_{12}} g_{(1)}(v_2)$ to $G_{in}^{i+1}(V_b, E_{in}^{i+1})$.
 - ii. If $v_1 = v_0$, add an edge $g_{(1)}(v_1) \xrightarrow{w_{12}+L} g_{(1)}(v_2)$ to $G_{in}^{i+1}(V_b, E_{in}^{i+1})$.
 - iii. If $v_2 = v_0$, add an edge $g_{(1)}(v_1) \xrightarrow{w_{12}-L} g_{(1)}(v_2)$ to $G_{in}^{i+1}(V_b, E_{in}^{i+1})$.
 - (g) Set $i = i + 1$.

At step 3 – (d) the graph $G_{in}^i(V_b, E_{in}^i)$ is returned. This graph can be shown to be equal to $G^{1,k}(f_N^2)$ [3]. Once we find homogeneous created edge sets on $V_{b,1} \cup \{v_0\}$ at step 3 – (d), asymptotic parametric bound functions for job start times can be found from the graph $G^{1,k}(f_N^2)$. From this graph the variables in the sequence $\langle f_N^2, s_N^2, \dots, f_1^2, s_1^2 \rangle$ are eliminated to obtain the parametric bound functions for each s_i^2 , $1 \leq i \leq N$. During this elimination process, the weights of edges connected to or from v_0 have to be modified appropriately to reflect scheduling window index $j \geq 2$ as well as the node index of the graph. For example,

- if an edge $v_0 \xrightarrow{w} s_i^2$ is obtained after eliminating $\langle f_N^2, s_N^2, \dots, f_i^2 \rangle$, then a formula $s_i^j \leq w + (j - 2)L$ must be used in deriving asymptotic parametric bound functions for s_i^j .

- if an edge $s_i^2 \xrightarrow{w} v_0$ is obtained after eliminating $\langle f_N^2, s_N^2, \dots, f_i^2 \rangle$, then a formula $-w + (j-2)L \leq s_i^j$ must be used in deriving asymptotic parametric bound functions for s_i^j .
- if an edge $s_a^1 \xrightarrow{w} s_i^2$, is obtained after eliminating $\langle f_N^2, s_N^2, \dots, f_i^2 \rangle$, then a formula $s_i^j - s_a^{j-1} \leq w$ must be used in deriving asymptotic parametric bound functions for s_i^j .

After obtaining asymptotic parametric bound functions for s_i^j , $2 \leq j$, we can also find parametric bound functions for Γ^1 by eliminating nodes from $G^{1,k}(f_N^1)$.

The time complexity of the off-line scheduler is $O(n^2N^3)$ where n is the number of jobs in a scheduling window that have relative timing constraints with jobs in the next scheduling window, and N is the number of jobs in each scheduling window [3].

B.5 On-line Dispatcher

The on-line dispatcher of the scheduler processes the information transferred to it from the off-line module, processes them to create the run-time data structures that are used in the process of determining the dispatch time for the different task instances. The Dispatcher can also determine the schedulability of a new aperiodic real-time task introduced to the system at run-time. This is done by moving task instances around in accordance with their parametric functions to preserve total schedulability. The algorithm to insert an aperiodic task at run-time is described in [3].

This section describes the data structures used by the on-line component, and then explain the use of these data structures to handle the task dispatching process.

B.5.1 Run-time data structures

Scheduling information needed for the dispatching process are transferred to the on-line component by means of a file written by the off-line component. This information include task descriptions, relative timing constraints in the form of parametric functions description used to determine the minimum and maximum bounds on the execution start times for the task instances. Run-time information is stored in the form of a calendar of the tasks and their timing properties. The Dynamic Calendar has two main components:

Dependency graph shown in Figure 16, it is represented as a list of tasks that are active at the current time each node in the list contains

- Task ID

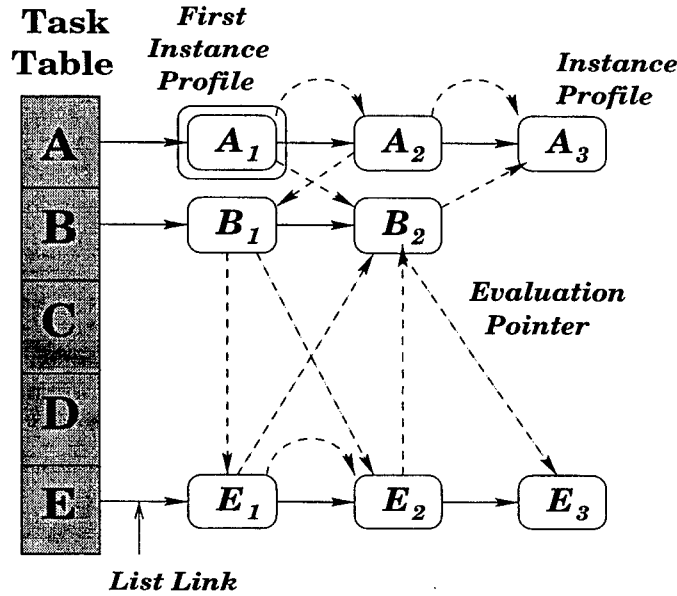


Figure 16: Dependency Graph

- A linked list of the task *instance profiles* for that task, each of the task-instance profiles contains the following information:
 - Instance ID
 - Maximum execution time (WCET)
 - *Activation counter* that describes the number of life cycles of the task that this instance is going to remain active in
 - *Instance functions*, a list of parametric functions, each containing a pointer to function code, a list of the function parameters, and an *Evaluation counter* for the unresolved parameters in the function.
 - *Result lists*, which are lists of pointers (*Evaluation pointers*) to the locations of parameters for the parametric function of other task instances, these pointers indicate that values from this task instance are the actual parameters for the formal parameters in the other task instance functions. A separate list is maintained for each value to be propagated.

Time ordered list A time ordered list of task instances is maintained by the run-time module, its entries represent task instances that the run-time module have full knowledge about their execution profile, that is the parameters to their parametric functions are all satisfied and the functions

are evaluated to yield an absolute time to start the execution of the task instance. Entries in the TOL consist of the absolute minimum and maximum times that this task instance can start its execution. It also include a pointer to the task instance profile in the dependency graph. Entries in this list are ordered according to their earliest starting times.

B.5.2 Run-time execution model

The dispatcher propagates parameters of the parametric functions, and dispatches the correct task instance according the calendar generated by the off-line component. The run-time module starts by processing the Calendar information passed by the off-line component in the form of parametric functions, the scheduling information is stored at run time in the dependency graph. The TOL is initialized with one task instance, which is the task marked by the off-line scheduler to be executed first, and its execution time is not dependent on time values generated by the other task instances. Task execution phase follows the procedure described next.

Dispatch the first task instance in the TOL, and start executing it in the earliest possible time between its minimum and maximum start times. The kernel schedules an interrupt at the end of the WCET of that task instance in order to be able to gain control and maintain the schedule of the remaining tasks execution.

After the current task instance finishes execution, kernel gains control again, it starts by propagating the timing information generated from the finished task instance to all the function parameters that are dependent on these values using the results lists of these values in the task instance profile. If the unresolved parameters counter in any one of these task instances reaches zero, this means that the parameters to its functions are all satisfied and functions can be evaluated at this point. The absolute boundaries on the starting times for these instances are calculated, and the instances are inserted in the TOL, their counters are reset to their original values in the instance profiles. The dispatcher also maintains the information in the task-instance profiles regarding the number of cycles the instance is going to be active in, this counter is decremented every time the instance is executed. If this counter was initialized with a negative value, this will cause the dispatcher to run this task periodically for as long as the operating system kernel is running this particular application. The on line dispatcher time complexity is $O(N)$.

The main steps for the On-line dispatcher is shown in the following algorithm.

Algorithm 4 *On-Line Dispatcher.*

1. Load the dynamic calendar by tasks parametric functions.
2. Insert the first task instance in the TOL.
3. while (TOL not empty) {
 - (a) Get first task instance in TOL (I_{top}).
 - (b) Schedule a time interrupt to occur immediately after $s_{top} + WCET(I_{top})$.
 - (c) Yield control to I_{top} .
 - (d) When I_{top} finishes or the scheduled interrupt occurs
 - Stop the execution of I_{top} if it is still running.
 - Record its finishing time $f_{I_{top}}$.
 - Substitute the start time s_{top} in all items in its evaluation list.
 - Decrement the evaluation counters of all the elements on the evaluation list of s_{top} .
 - Substitute the finish time f_{top} in all items in its evaluation list.
 - Decrement the evaluation counters of all the elements on the evaluation list of f_{top} .
 - If the evaluation counters of any instance reaches zero, then
 - insert this instance in TOL.
 - Decrement its activation counter by 1, if it reaches 0, the instance is removed from the dependency graph.
 - Restore all its evaluation counters to their initial values.}

B.6 Conclusion

In this paper, a new model is developed for dynamic time-based scheduling scheme. Using this scheduling model, it is possible to schedule more general tasks such as periodic, or aperiodic. Tasks can execute in any general pattern other than strict periodic, for instance the system can schedule periodic tasks with variable inter-instance periods. The tasks scheduled by the system at pre-runtime must show a repeated pattern in order for the scheduler to be able to constitute the scheduling windows of the given tasks. The timing constraints satisfiable by the system include the following:

- Ready time
- Deadline time
- Communication constraints

- Mutual exclusion constraints
- Together constrains
- Relative timing constraints

Some of the benefits that can be achieved by the given model can be summarized in the following main points:

- Ability to add aperiodic tasks at run-time.
- Ability to schedule more general tasks.
- Variation of the run-time behavior depending on values generated by executing tasks, or system state to change the parametric functions calculated by the off-line component at pre-runtime.
- Using parametric function makes use of the slack time to run non-real-time tasks, or to finish the schedule as early as feasible.

The proposed model also gives some possibilities of fault tolerance by allowing the operating system kernel to gain control, or update the different functions parameters in case of failure. Some of the fault tolerance abilities that are given by the system are:

- Substitution of minimum values for parameters in case of failure of task instances generating the parameter value to keep the feasible total schedule.
- Using the maximum execution time for the task instances to generate a time interrupt, should the task instance execute more than the max time allowed for it.

As *future extensions* to this model, different things can be further extended to give more generality and flexibility in the scheduling capabilities of the system. Some of these extensions are:

- The dynamic calendar functions can be parameterized by values other than just the start or finish times of previous tasks, such as system state variables. This can give more capabilities to support inter-task dependencies, and fault tolerance.
- The scheduling model can be extended to support multi-processor systems. To do this, several issues have to be considered, such as what kind of information have to be sent out to other nodes, and how parametric functions can be found.

- In this paper it is also assumed that a total ordering among tasks is found at pre-runtime by an off-line scheduler. Previous work by Cheng *et al.* [2] and Mok *et al.* [9] use a heuristic approach called *smallest latest start time first* to schedule task instances with relative constraints. However, their heuristics don't fully reflect the relative timing constraints. Improved heuristic functions may be developed if the constraint graph structure is utilized.
- The total order and dynamic calendar were both calculated at pre-runtime. Aperiodic real-time tasks can also be added to the dynamic calendar at run-time [3]. The addition of hard real-time periodic tasks at run-time may be further studied to find the way parametric functions are to be changed in that case.

References

- [1] C. Hou C. Han and K. Lin. Distance-constrained scheduling and its applications to real-time systems. *IEEE Transactions on Computers*, To Appear.
- [2] S. Cheng and A. K. Agrawala. Scheduling of periodic tasks with relative timing constraints. Technical report, CS-TR-3392, UMIACS-TR-94-135, Department of Computer Science, University of Maryland, December 1994.
- [3] Seonho Choi. *Dynamic Time-Based Scheduling for Hard Real-Time Systems*. PhD thesis, University of Maryland at College Park, 1997.
- [4] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.
- [5] G. Fohler and C. Koza. *Heuristic Scheduling for Distributed Real-Time Systems*. PhD thesis, Technische Universitat Wien, Vienna, Austria, April 1989.
- [6] D. K. Hammer J. P. C. Verhoosel, E. J. Luit and E. Jansen. A static scheduling algorithm for distributed hard real-time systems. *Real-Time Systems*, 6(2):227–246, 1991.
- [7] A. Burns K. Tindell and A. Willings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2), March 1994.
- [8] M. H. Klein M. G. Harbour and J. P. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

- [9] Aloysius K. Mok and Duu-Chung Tsou. The msp.rtl real-time scheduler synthesis tool. In *IEEE Real-Time Systems Symposium*, pages 118-128, December 1996.
- [10] W. Pugh R. Gerber and S. Saksena. Parametric dispatching of hard real-time tasks. *IEEE Transactions on Computers*, 44(3), March 1995.
- [11] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *10th International Conference on Distributed Computing Systems*, pages 108-115, 1990.
- [12] Manas Saksena. *Parametric Scheduling for Hard Real-Time Systems*. PhD thesis, University of Maryland at College Park, 1994.
- [13] Manas Saksena. Design and implementation of maruti ii. Technical report, University of Maryland at College Park, 1995.
- [14] T. Shepard and J. A. M. Gagne. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software Engineering*, 17(7):669-677, 1991.
- [15] Bao Trinh. Creating a mruti executable. Technical report, Systems Design and Analysis Group, Department of Computer Science, University of Maryland at College Park, April 1997.
- [16] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, SE-16(3):360-369, 1990.